Grounding Linguistic Analysis in Control Applications

by

Satchuthananthavale Rasiah Kuhan Branavan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Electrical Engineering and Computer Science
February 17, 2012

Certified by .....
Regina Barzilay
Associate Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor
Chair of the Committee on Graduate Students

Grounding Linguistic Analysis in Control Applications

by

Satchuthananthavale Rasiah Kuhan Branavan

Submitted to the Department of Electrical Engineering and Computer Science
on February 17, 2012, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis addresses the problem of grounding linguistic analysis in control applications, such as automated maintenance of computers and game playing. We assume access to natural language documents that describe the desired behavior of a control algorithm, either via explicit step-by-step instructions, via high-level strategy advice, or by specifying the dynamics of the control domain. Our goal is to develop techniques for automatically interpreting such documents, and leveraging the textual information to effectively guide control actions.

We show that in this setting, langauge analysis can be learnt effectively via feedback signals inherent to the control application, obviating the need for manual annotations. Moreover we demonstrate how information automatically acquired from text can be used to improve the performance of the target control application.

We apply our ideas to three applications of increasing linguistic and control complexity – interpreting step-by-step instructions into commands in a graphical user interface; interpreting high-level strategic advice to play a complex strategy game; and leveraging text descriptions of world dynamics to guide high-level planning. In all cases, our methods produce text analyses that agree with human notions of correctness, while yielding significant improvements over strong text-unaware methods in the target control application.

Thesis Supervisor: Regina Barzilay
Title: Associate Professor

Acknowledgments

This thesis is the culmination of an exciting six year adventure which started with the insistent, but at the time unfounded, conviction that *there must be a better way for people to configure and control software systems without having to write messy code.* Six intellectually invigorating years later, that dream, I believe, is within grasp. For this I'm forever grateful to my advisor Regina Barzilay, who had the vision to see the merits of an initially incoherent idea, and the perspicacity to know when we were on the right research path – I could not have asked for a better advisor.

I have been very fortunate in having had a fantastic thesis committee. The advice and guidance of both Leslie Kaelbling and Dina Katabi had a direct impact on this work from the very beginning, well before I asked them to be readers on my committee. In fact, I have to thank Dina Katabi, Nate Kushman and Martin Rinard for insisting that *if we want to do automatic instruction interpretation, we must use Microsoft Windows as the target environment* – this insistence ended up setting the fundamental direction for the methods developed here.

This thesis would not have been possible without the many wonderful people I have had the good fortune of working with at MIT. This work is the direct result of collaborations with Harr Chen, Nate Kushman, Tao Lei, David Silver, Luke Zettlemoyer, and my advisor Regina Barzilay. The conversations, ideas, feedback and assistance from many others at MIT has also been invaluable – Yevgeni Berzak, Mike Collins, Pawan Deshpande, Jacob Eisenstein, Tommi Jaakkola, Yoong Keok Lee, Tahira Nasim, Christina Sauper and Benjamin Snyder. A special thanks to Marcia Davidson for her help with all things administrative, and the impromptu conversations about life outside research.

I'm indebted to Indra Dayawanse of the University of Moratuwa, Samanthie Gunasekara, Ajit Samaranayake and Hemantha Jayawardana of Millennium IT, and Saman Amarasinghe of MIT all of who, along with my family, were instrumental in my decision to return to graduate school.

I dedicate this thesis to my wonderful family: My parents and sisters – who

have been my first and best mentors to science and its philosophy – Vimaladevi, Satchuthananthavale, Vithiyani and Imasala; my brother-in-law, nephew and niece, Kumaradevan, Kulaghan and Kauline; and my many uncles, aunts and cousins.

Bibliographic Note

Portions of this thesis are based on prior peer-reviews publications. The instruction interpreting work of Chapter 2 was originally published in Branavan et al. [10] and Branavan et al. [11]. The work on interpreting strategy guides from Chapter 3 was first published in Branavan et al. [12] and Branavan et al. [13]. An expanded version of this work is pending publication. Finally, a version of the method for learning high-level planning from text (Chapter 4) is currently under peer-review.

The code and data for the methods presented in this thesis are available at `http://groups.csail.mit.edu/rbg/code/grounding_language_in_control`.

Contents

List of Figures

List of Tables

1

Introduction

Natural languages are the medium in which the majority of humanity's collective knowledge is recorded and communicated. If machines were able to automatically access and leverage this information, they could effectively perform many tasks that are currently considered to be computationally intractable, thus requiring human involvement. For example, computers could maintain themselves by reading help documents or solve hard planning tasks by acquiring relevant domain knowledge from text. Today, the only way to infuse such human knowledge into computational algorithms is to have humans in the loop – i.e., to manually encode the knowledge into heuristics, through annotations, or directly into the model structure itself. Our ultimate goal is to automate this process, so that machines can access required knowledge directly from text. One path to this goal is to perform a semantic interpretation of text by grounding the textual information in the objects, actions and dynamics of the physical world. As a step towards this goal, this thesis looks at the connection between control applications and the semantics of language.

From a linguistic viewpoint, the grounding of language in control applications presents a very natural notion of language semantics. Today in the field of natural language processing, a plethora of semantic annotation schemes are in use, most of them based on linguistic notions of semantics [78, 43, 26, 80]. However, there is no empirical evidence or consensus on which annotation scheme is good in terms of real-world applicability. The connection between language and control applications, however, provides a new perspective on language semantics. In this context, semantics

The natural resources available where a population settles affects its ability to produce food and goods. **Cities built** on or near **water** sources can **irrigate** to increase their **crop** yields, and **cities** near **mineral** resources can **mine** for **raw materials**. **Build** your **city** on **plains** or **grassland** with a **river** running through it if possible.

Figure 1-1: An extract from a document describing the dynamics of a complex strategy game. Words that denote objects or actions in the game are highlighted in bold.

serves as the bridge between text and the control application. This allows us to define the representation of semantics with respect to the control application, and avoid imposing subjective human notions of correctness.

In this dissertation, I explore two aspects of the connection between language and control applications: first, how the semantic analysis of language can be driven by control performance; and second, how information from text can be leveraged to improve performance in complex control systems. These two aspects are in fact complementary, and language analysis is central to them both. While addressing these aspects jointly is particularly challenging, doing so enables a novel source of supervision for learning language analysis. Since we assume that text contains information useful for the target control task, correctly interpreting the text must, by definition, improve control performance. Thus, if the performance of the control application itself can be measured, this measurement can serve as a feedback signal for learning language analysis. This is the key idea behind this thesis – that by connecting the semantic analysis of language to control applications, we can leverage the synergy between the two to simultaneously learn both language analysis and application control with little or no prior knowledge.

As a motivating example for our language interpretation task, consider the text shown in Figure 1-1. This is an extract from a user manual describing the dynamics of a complex strategy game. This text is written to help human players learn the game, and provides useful information about game dynamics – for example, that *crop* yield can be increased by *irrigating* land. Given this text, we wish to identify the objects and actions that are denoted by words in the text, such as "irrigate", "city", and

"water". Furthermore, we also want to extract the relationships described in the text – e.g., that *water* is a prerequisite for *irrigation.* This grounding of objects, actions and relations can be acquired by an automated agent the hard way – i.e., by acting in the world and observing how specific actions change the world state. In a complex world having a large state space, this approach can be prohibitively expensive. The ability to interpret the text in Figure 1-1, however, can allow the agent to directly leverage the information from text, and significantly improve its ability to play the game.

Inducing useful domain knowledge from text, however, poses several challenges beyond the quintessential issues of linguistic variability and ambiguity:

- **Situational Relevance** The first challenge is to identify *situationally relevant* text – i.e., textual information that is useful in the current state of the world. This is crucial since text documents often include irrelevant background material, and even text that contains useful information may be valid only in specific scenarios. For example, a player trying to increase crop yield should ignore the information about *mining* and *minerals* in Figure 1-1.

- **Abstraction Level** The second challenge is the level of abstraction at which text refers to the world. At the lowest level of abstraction, words in the text will describe directly observable attributes of the world – e.g., words such as "city" and "irrigate" which denote an object and an action respectively. Often, the text will also describe abstract attributes of the world – such as the relationship between different objects. An example of this from Figure 1-1 is the description of the precondition relationship between *water* and *irrigation.* Such *abstract groundings* are particularly challenging to learn since the corresponding abstract world attribute also needs to be learnt.

- **Incomplete Textual Information** The third challenge we need to address arises from the essentially incomplete nature of textual information – i.e., text will not necessarily provide all of the information required for effective control. For example, in Figure 1-1, the text recommends that cities be built on

grassland or plains, but does not describe the consequences of building a city near the sea. Hence, an algorithm that aims to perform effective control has to fuse information from both the text and the control application to address the information deficiencies of text.

To address these challenges of grounding language in control, we leverage feedback signals inherent to control applications as the supervision for learning. In particular, we assume that we have access to a noisy, real-valued reward signal that correlates with the quality of control actions. To learn effectively from such feedback in a principled fashion, we formulate our language grounding task as a *Markov Decision Process* (MDP) in the *Reinforcement Learning* framework [68]. The goal in this framework is to learn an optimal policy $p(a \mid s)$ for selecting actions a, that when executed from state s will maximize the expected future reward. Our novel formulation for text interpretation defines MDP actions a in terms of both text analysis decisions and control actions, while combining text and control information in the MDP state s. This formulation enables language analysis to be learnt from control feedback, while at the same time allowing control actions to be guided by textual information. Our algorithms explicitly model abstraction, situational-relevance, and world dynamics to enable effective analysis of complex language.

We develop these ideas in the context of three different control applications:

1. **Interpreting Imperative Instructions** The text provides step-by-step instructions specifying the sequence of commands that need to be executed in the world. The entire text is assumed to be relevant to the task. Grounding happens at the lowest level of abstraction – i.e., the text describes concrete objects and actions in the world. The challenge of this application arises from the fact that the text instructions are the *only* definition of the target control task. Hence, the control task cannot be completed without correctly interpreting the text.

2. **Interpreting Strategy Descriptions** The text provides situation-specific advice and general background knowledge about the world. While this advice

describes specific actions, and the situations in which those actions are useful, the interpretation algorithm needs to decide when to apply any particular action in the world. Moreover, the text contains significant amounts of background information that is not relevant to action selection. As in the first case, grounding is specified at the level of objects and actions. However, unlike in the first application, here we assume we are given a non-linguistic definition of the control task. The control task can hence be completed without interpreting the text. However, due to the complexity of the task, the search space is exponentially large, making domain knowledge extracted from language particularly crucial. The challenge here is to identify and extract useful information from text, while also allowing for effective control when such information is not available.

3. **Using Text to Guide High-level Planning** In contrast to both the above applications, the text here specifies only the dynamics of the world in general, and does not provide any advice about actions in the world. In particular, the text provides information about prerequisite relationships between objects in the world. Given this characteristic of the text, grounding is at the level of abstract relationships between objects. As in the second application, the control task here is defined independently of text, and all of the same challenges also apply. The primary complexity of this application, however, lies in the difference in abstraction level between the high-level relation descriptions in text, and the concrete objects in the world.

I summarize these three control applications below, describing our approach to effective language analysis in each case.

1.1 Interpreting Imperative Instructions

Our first application is to translate documents containing step-by-step instructions into sequences of commands in a given world. In this scenario we assume that the entire text of the document is useful for the application, and that grounding occurs at the level of concrete objects and commands. An example of this application is shown in Figure 1-2, where the text contains instructions for configuring a specific service in the Microsoft Windows operating system, and we wish to execute the corresponding commands in the actual OS. While this is the simplest of the three scenarios discussed in this thesis, it is also the most unforgiving in terms of language analysis. The commands to be executed in the OS are specified only via the instruction text, making the control task impossible to complete without correctly interpreting the text.

The unforgiving nature of this task is also an opportunity for learning via trial and error. Specifically, we formalize our task of mapping instructions into commands in the reinforcement learning framework, where the goal is to learn an action-selection policy. We define an action as the selection and translation of a word span from the text into commands in the target environment, thus enabling our method to learn text interpretation. As the reward signal for learning, we define an intuitive albeit noisy feedback signal that measures the quality of the mapped command sequence. This reward signal emulates the way in which a typical lay human would identify that they had made a mistake while following an instruction document – i.e., by testing whether the remaining instruction text matches any text labels in the environment.

This reinforcement learning approach to instruction interpretation allows us to bypass the traditional reliance on manual annotations, while achieving a interpretation performance surprisingly equal to that of an equivalent supervised technique.

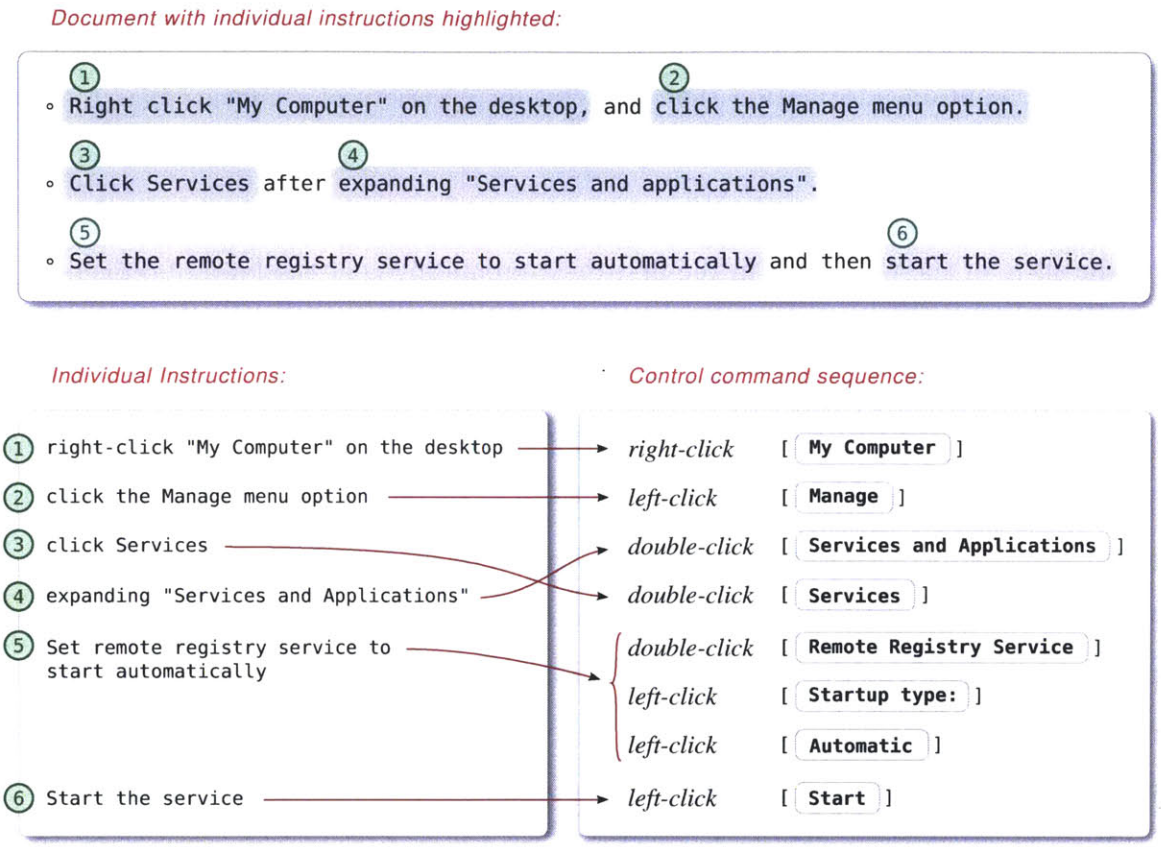


Figure 1-2: An example interpretation of instructions from a Microsoft Windows help document into GUI commands. The text specifies the sequence of commands that need to be executed, and the GUI object for each command. Note that it is the document itself that defines the task to be performed in the OS. Hence, the task cannot be completed without correctly interpreting the text.

1.2 Interpreting Strategy Descriptions

Our second task is to interpret strategy documents that contain general advice on how to behave effectively in the world. We aim to use the domain knowledge extracted from text to control the actions of an agent in the given world. Unlike in the first task, the documents we hope to interpret do not provide step-by-step instructions, but rather contain information that is useful for a variety of scenarios that may be encountered in the world. In addition, we assume that only a fraction of the document contains useful information, and that the relevance of even this information depends on the state of the world. As in the first task, grounding is at the level of objects and actions.

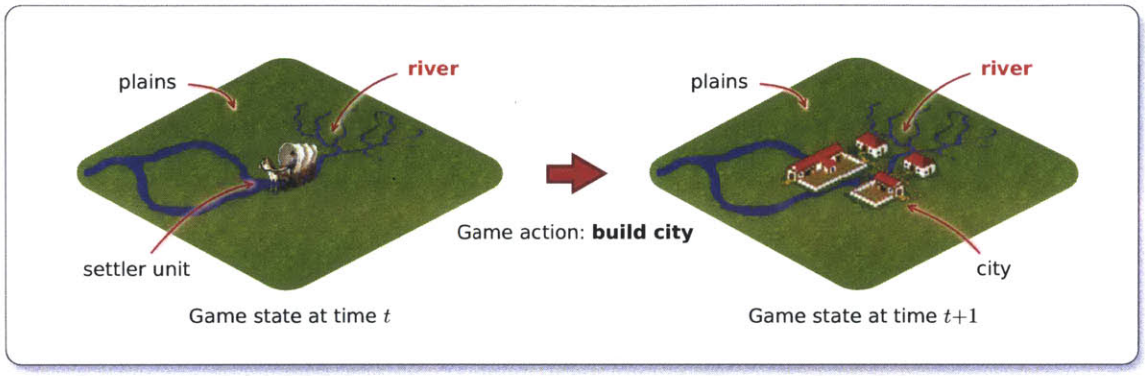
Figure 1-3 shows an excerpt from a document with these characteristics – a user guide for the strategy game *Civilization II*. This text describes game locations where a particular game action can be effectively applied, but it leaves all other details about that action unspecified. Any algorithm that aims to leverage such texts has to therefore overcome two specific challenges: first, it needs to learn to identify the portions of text that are relevant to the current world state; second the algorithm needs to effectively fuse knowledge extracted from text with information from the world to compensate for deficiencies in the text.

We address these challenges by explicitly modeling the relevance of textual information, and by jointly modeling both linguistic and control decisions. We encode this joint model in a multi-layer neural network that is learnt by interacting with the world in the Monte-Carlo Search framework. The Linguistic decisions of text relevance and grounding are modeled by the hidden layers of this network. We test our method on the complex strategy game of Civilization II, using the game's official manual as the source of textual information. As shown by our results in Chapter 3, our linguistically-informed method significantly outperforms a strong baseline that does not have access to the text. Furthermore, while being learnt only from world-feedback, the text analysis produced by our method conforms well with common notions of linguistic correctness.

Excerpt from strategy document:

The natural resources available where a population settles affects its ability to produce food and goods. Cities built on or near water sources can irrigate to increase their crop yields, and cities near mineral resources can mine for raw materials. Build your city on plains or grassland with a river running through it if possible.

Scenario 1: ***Building a city near a river***



Scenario 2: ***Building a city near the sea***

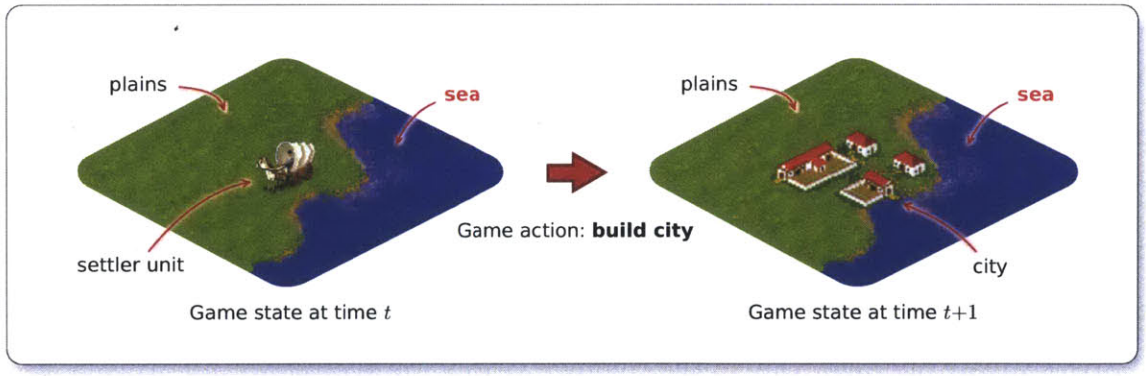


Figure 1-3: An excerpt from the user manual of the strategy game *Civilization II*, and two game scenarios highlighting the challenges of interpreting such text. The scenarios show the game action *build city* being executed by a *settler* in two different game states. In the first scenario, the city is being built on a *plain* near a *river* as advised by the final sentence of the text. If a game playing algorithm can identify this sentence as relevant to the game state, and ignore the remaining text, it can leverage the text information to play better. The second scenario, however, is not covered by the manual – which does not provide any information about cities near the sea. Thus, to be effective, an algorithm for leveraging text needs to also use information from the world to compensate for gaps in the text.

1.3 Using Text to Guide High-level Planning

Our final task is to leverage textual information about the preconditions and effects of actions in the world to perform effective high-level planning in a complex domain. The unique challenge of this task is the mismatch in the level of abstraction between information provided in the text, and the granularity of planning primitives in the world. Consider for example, the text shown in Figure 1-4. This text describes a precondition relation between two objects in a virtual world – i.e., that seeds are required to grow wheat. The corresponding plan connecting these to objects, however, is at a level of abstraction far lower than that of the text.

We address the challenge of the mismatch in abstraction granularity by learning a model of the world that explicitly encodes the abstract concepts described in the text. This allows us to model language grounding at the level of relations, in contrast to prior work which focused on object-level grounding [52, 66, 79, 27, 50, 49, 10, 74]. We implement our idea in a high-level planning algorithm that jointly learns to predict precondition relations from text and to perform planning guided by those relations. Our method uses a standard low-level planner to convert high-level plans to actions in the world. The success or failure of the low-level planner is then used as the supervision signal for our model.

We evaluate our algorithm in the complex virtual world of *Minecraft*, using a large collection of user-generated on-line documents as our source of textual information. While using planning feedback as its only source of supervision, our method is able to predict preconditions from text as well as a supervised SVM baseline. Furthermore, our algorithm is able to leverage this precondition information to significantly improve planning performance over text-unaware algorithms.

Text:

Seeds planted in **farmland** will grow to become **wheat** which can be harvested

Precondition Relations extracted from text:

seeds ⟶ wheat

farmland ⟶ wheat

Plan corresponding to the task of "growing wheat"

1. pickup tool: **shears**
2. collect **seeds** from **tallgrass** using **shears**
3. pickup tool: **hoe**
4. plow **land** with **hoe** at (2,0) into **farmland**
5. plant **seeds** at coordinates (2,0)
6. fertilize **seeds** at (2,0) with **bonemeal**
7. wait for **wheat** to grow
8. pickup tool: **shears**
9. harvest **wheat** with **shears** at (2,0)

Graphical representation of plan showing grounding states for precondition relations:

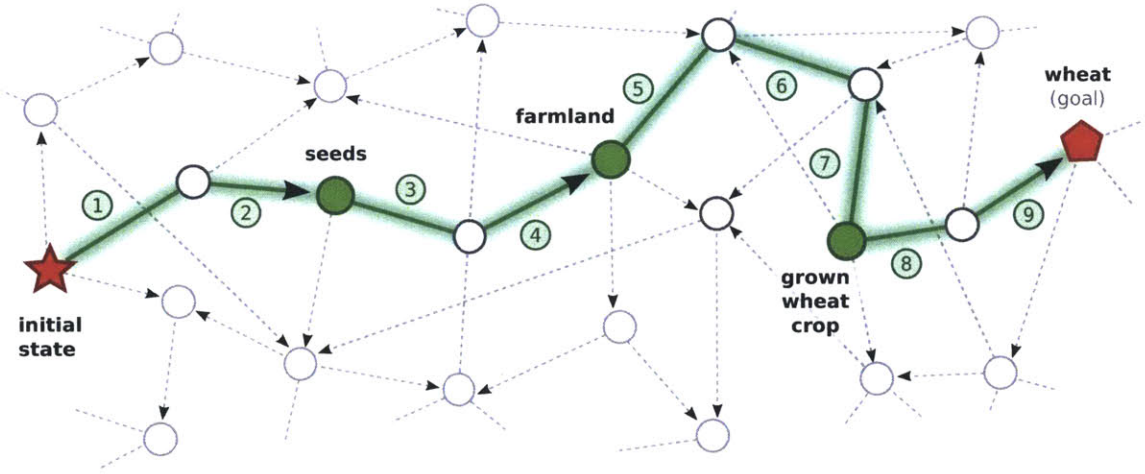


Figure 1-4: An excerpt from a help document for the virtual world of *Minecraft* describing precondition relationships between objects – i.e., that both *seeds* and *farmland* are required to grow *wheat*. Note the difference in abstraction level between these precondition descriptions and an actual executable low-level plan to grow wheat. In fact multiple low-level actions are necessary to connect the objects in the precondition relationships. Language grounding in this scenario requires our method to also learn the abstract relationships between world objects, thus providing a *grounding point* for language.

1.4 Contributions

The primary contributions of this work are twofold:

- **Learning language from control feedback** We show that given interactive access to a world, and text that provides useful information about that world, it is possible to learn effective text analysis without any manually annotated data. The performance of our methods rivals that of equivalent supervised techniques on a variety text analysis tasks.

- **Guiding control actions using textual information** We demonstrate the feasibility and effectiveness of automatically leveraging information from natural language texts to guide complex control actions in the world. As shown by our results, our approaches are able to significantly improve control performance using textual information.

Starting with little or no prior knowledge about either language or the control application, our methods are able to achieve both of the above results in a compelling manner – learning language analysis, and effective control behaviour. Additionally, the language grounding models described in this thesis are themselves important contributions. In particular, we show how situationally-relevant text and abstract relations from text can be modeled for effective grounding and control. These modeling techniques open the way for the automatic interpretation of text containing even more complex linguistic phenomena.

1.5 Outline

The remainder of this thesis is organized as follows:

- **Chapter 2** discusses the interpretation of imperative, step-by-step instructions for performing tasks in the world, and presents our approach for learning to map such documents into actions.
- **Chapter 3** describes our method for interpreting high-level strategy guides which contain situationally-relevant advice.
- **Chapter 4** presents our model for automatically extracting precondition relations from text, and inducing high-level plans based on the extracted information.
- **Chapter 5** concludes the thesis with a summary of the main points, and directions for future work.

2

Interpreting Instructions into Actions

In this chapter, we consider the task of automatically interpreting natural language instructions into executable commands. Using a reinforcement learning approach, we show that this task can be learnt by executing the interpreted commands in a target environment and observing the results. Our method, which requires no manual supervision, performs as well as equivalent supervised approaches. We further demonstrate that learning a model of the target environment allows us to handle high-level instructions, which assume prior knowledge on the part of the reader, and omit some of the details necessary for correct interpretation.

2.1 Introduction

The ability to automatically map natural language instructions into executable commands would enable the automation of a wide variety of tasks that currently require direct human involvement. Examples of such tasks include configuring and maintaining software systems based on how-to guides, and controlling robots using instruction manuals. This natural language interpretation task has been widely studied from the early days of artificial intelligence [76, 24]. However, prior attempts have relied either on human specified rules or on learning from manual annotations, restricting their ability to scale. Our approach removes this limitation by learning instruction interpretation without the need for annotated data. We rely on the observation that in many applications, the validity of an instruction mapping can be verified by executing

the induced command sequence in the corresponding environment and observing the resulting effects. Operating in a reinforcement learning framework, our method learns by leveraging this feedback, and performs as well as manually supervised techniques.

For concreteness, consider the text from a Microsoft Windows troubleshooting guide shown in Figure 2-1. This document describes how to configure the "remote registry service" so that it starts automatically. Our goal is to map this text into the corresponding sequence of eight GUI commands that correctly perform the configuration task. Being written for a human audience, these kinds of documents generally assume certain prior knowledge on the part of the reader. As such they often gloss over low-level details, and succinctly describe multiple commands in a single instruction. An example of such a *high-level* instruction can be seen in step five of Figure 2-1.

In this chapter, we assume that each document is composed of a sequence of instructions, each of which can take one of two forms:

- **Low-level instructions**: these explicitly describe single commands. E.g., the third instruction, "Click Services" in Figure 2-1. Note that low-level instructions fully specify the command to executed, along with the parameters of that command. As such, the information available from text is sufficient to correctly select and execute the environment command.

- **High-level instructions**: these describe a sequence of one or more environment commands, which are not explicitly described by the instruction. E.g., instruction five in Figure 2-1 – "Set the remote registry service to start automatically". Note high-level instructions do not specify the details of every command in the corresponding sequence. For this reason, the correct command sequence cannot necessarily be induced based only on textual information. Effectively collecting and using information about world dynamics, in addition to the text, thus becomes crucial when interpreting these instructions.

The process of interpreting text documents containing such instructions involves several challenges:

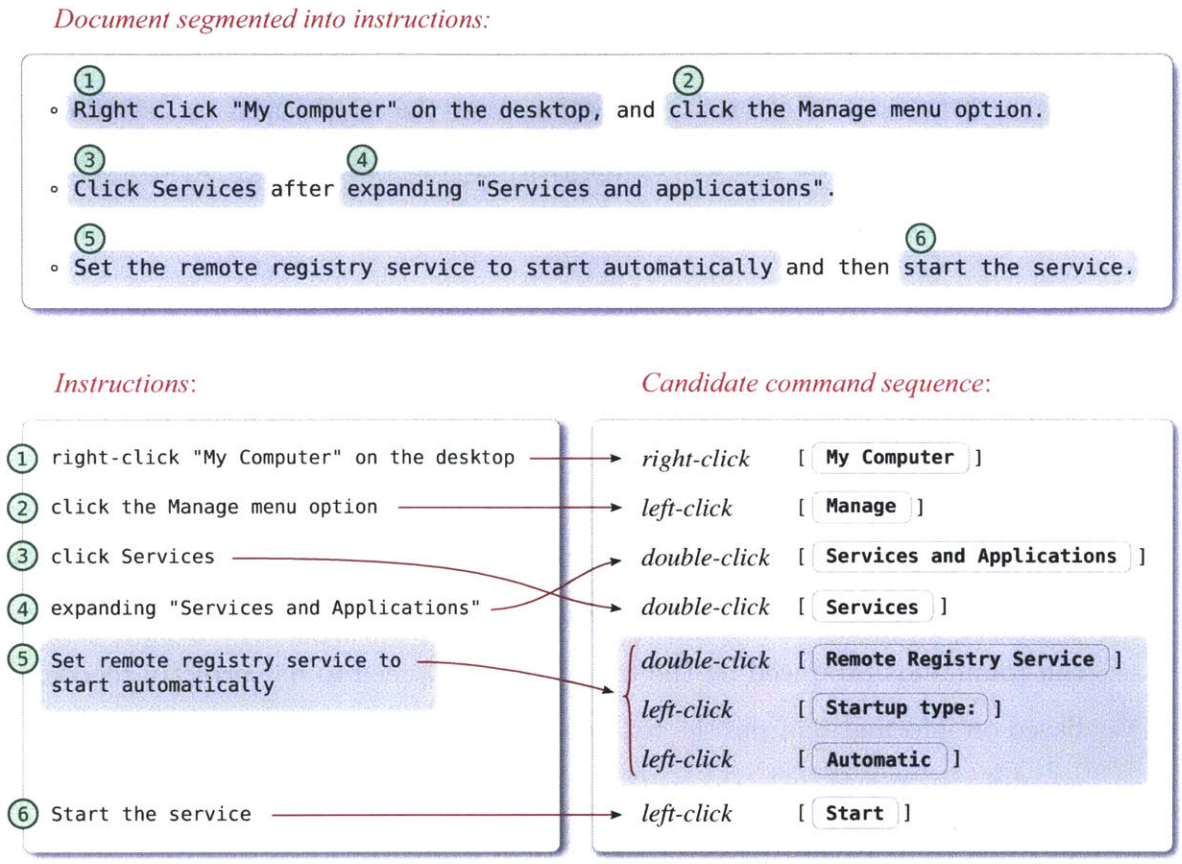


Figure 2-1: An example mapping of a document into a command sequence. Since a single sentence can describe multiple instructions, the first step as shown on top, is to segment the text into individual instructions. Each instruction then needs to be translated into the corresponding GUI commands. An instruction that maps to a single GUI command is termed a *low-level instruction* – e.g., instructions one through four. *High-level instructions* are those that map to a sequence of commands – e.g., instruction five. Notice that the execution order of commands can be different from the order of instructions in the text. Thus the mapping process also has to handle command reordering when necessary.

- **Segmentation** As can be seen from the text in Figure 2-1, a single sentence can describe multiple instructions. Thus, the first challenge is the linear segmentation of each sentence into sets of word spans, with each span specifying a single instruction. For example, the sentences in Figure 2-1 segment into two instructions each. Note that we assume that the words used to describe two different instructions will not be interleaved, and that a single instruction will not span multiple sentences.

- **Translation** The second challenge is to translate each low-level instruction into its corresponding GUI command, and each high-level instruction into the command sequence it describes. Figure 2-1 shows this translation step for each instruction.

- **Reordering** The order in which commands are described in the text may not always match the order in which they need to be executed. In such cases, the commands produced by the translation step need to be correctly reordered based on information from the text and the state of the target environment. For example, in Figure 2-1, the order of instructions three and four are switched between the text and the correct execution sequence.

- **World dynamics** As mentioned before, in the case of high-level instructions, the text does not fully specify every command in the corresponding sequence. To correctly handle these instructions, an interpretation algorithm needs to compensate for the information deficiency of text using knowledge about world dynamics. The challenge here is to automatically and efficiently acquire useful information about the world, and effectively incorporate this information in the interpretation process. For example, in instruction five from Figure 2-1, the algorithm needs to know that in the service management application of Microsoft Windows, *left clicking* on the *Startup type* widget will allow it to configure a service to start automatically.

Our aim is to address the above challenges and learn instruction interpretation without relying on human annotations. To achieve this goal, we first need an alternate

source of effective supervision. Conveniently, in many applications, the validity of an instruction mapping can be verified by executing the induced action sequence in the corresponding environment and observing the resulting effects. For instance, in the example from Figure 2-1, we can assess whether the goal described in the instructions is achieved – i.e., whether the *remote registry service* starts automatically. The key idea of our approach is to leverage such a validation process as the primary source of supervision to guide language learning. This form of supervision allows us to learn text interpretation in scenarios where standard supervised techniques are not easy to apply – such as for example, when human annotations are either not available, or difficult to create.

To address the challenges of learning instruction mapping from environment feedback, we formulate our task as a reinforcement learning (RL) problem. RL is a well studied and principled framework for learning models via validation from an environment [68]. We formulate our text interpretation model as a log-linear policy in this framework, encoding a variety of features from both the text and the environment. During learning, our method repeatedly constructs action sequences for a given set of documents, executes those actions, and observes the resulting environment feedback. The reward computed from this feedback is then used to update model parameters. In addition, our method also progressively builds a model of world dynamics based on the environment observations, thus enabling the interpretation of high-level instructions.

We evaluate our methods in two separate scenarios – the first where each document contains only low-level instructions, and the second where documents contain both high-level and low-level instructions. As the primary test domain for both scenarios, we use Windows troubleshooting guides from Microsoft's help website,[1] interpreting the text into GUI commands. The key findings of our experiments are twofold. First, our model trained using only simple reward signals is able to achieve surprisingly high results. It interprets low-level instructions with an accuracy of 79%, a performance

[1] http://support.microsoft.com/

that is on par with that of an equivalent supervised algorithm. Second, learning a model of the world allows our method to accurately interpret 62% of high-level instructions, compared to just 2% for a baseline that does not utilize an environment model.

The remainder of this chapter is structured as follows. We first describe prior work on grounded language acquisition, instruction interpretation, and reinforcement learning in Section 2.2. Section 2.3 details the formulation and structure of our model for instruction interpretation, and describes how we estimate its parameters using environment feedback. The following two sections, 2.4 and 2.5 discuss the application of our method to our two test domains, including our evaluation methodology. We present our experimental results in Section 2.6 before concluding in Section 2.7.

2.2 Related Work

In this section, we first discuss prior work in the areas of grounded language acquisition and instruction interpretation. Thereafter we look at related work in the field of reinforcement learning, focusing on its application to natural language processing in general, and our task in particular.

2.2.1 Grounded Language Learning

Our work fits into a broad class of methods that aim to learn language from a situated context [52, 3, 66, 54, 79, 17, 80, 43, 74, 71, 18]. These *grounded language learning* approaches aim to leverage non-linguistic information from a situated context as their primary source of supervision. While this non-linguistic signal may be noisy, it can often provide sufficient supervision to reduce or even obviate the need for manual annotations. Despite differences in the tasks to which they are applied, approaches to language grounding can be analyzed in terms of several shared characteristics.

- **Source of Supervision.** Prior work in language grounding has primarily operated on parallel corpora of text and grounding contexts. In the domains to which these methods have been applied, the natural language text is tightly and directly linked to the grounding context, allowing the methods to use the parallelism in the data to learn language analysis. For example, both Roy and Pentland [54] and Yu and Ballard [79] learn object names based on images paired with corresponding language. An alternative line of work has leveraged parallel data to ground language in a dynamic context, learning associations between action sequences and their textual descriptions. In this setup, for example, Fleischman and Roy [27] learn the grounding of text instructions to action sequences based on pairs of individual instructions aligned to the corresponding action sequence. Chen and Mooney [17], on the other hand, learn to ground a sports commentary to game actions from parallel, but unaligned data. In a similar vein, but operating in a real-world environment, Tellex et al. [71] learn to map human instructions into a sequence of actions by a physical robot.

While our work also aims to ground language in the context of an environment, in contrast to prior approaches, we do not assume access to any form of parallel data. Instead, our methods are allowed access to the target environment, and need to proactively interact with it to collect their training data. In complex domains, such as ours, the quality of information collected via interaction is critically important to learning. We address this challenge in a principled fashion by formulating our task in the reinforcement learning framework [68].

This approach of learning language analysis based on a feedback signal inherent to a given task has since been applied to a variety of applications. For example, interpreting navigation instructions [74], semantic parsing [19], and inducing semantic knowledge about a domain based on text [32].

- **Grounding Context.** Another feature common to traditional methods of grounding is the granularity of the non-linguistic structures to which the language is grounded. Specifically, in most prior work, language has been grounded to individual non-linguistic primitives – e.g., the visual representation of a single object [54, 79], individual actions [17], or individual semantic frames extracted from an observed action sequence [27]. In this setting, each instance of language grounding is independent of all other groundings. Even in cases where the text is grounded to a structured semantic representation such as first-order logic equations, the grounding decisions are not all sequentially interdependent [80, 43]. Furthermore, the non-linguistic context to which the text is grounded is constant across all decisions.

  In contrast, our task is to map a document containing instructions into a sequence of inter-dependent actions. While the text in our target domains can be segmented into words describing single actions, the state of the target environment changes with each executed action. This means that the grounding context for a given text segment is not known before all instructions that come before it have been correctly interpreted and executed. For this reason, grounding decisions within a document are dependent on each other, making our task

significantly more difficult. However, this dependency between decisions can also be a particularly powerful source of supervision – i.e., since an incorrect action could make it impossible to execute a future action, if we encounter text that cannot be mapped to any actions, it may indicate that we mapped a previous instruction incorrectly. As described in Section 2.4, we leverage this insight as our source of supervision.

2.2.2 Instruction Interpretation

The automatic interpretation of natural language instructions into actions in a world has been studied from the early days of AI [76, 24, 46, 40, 71]. Initial attempts at the task, such as Winograd's seminal work [76], used rule-based techniques and were limited to instructions dealing with a simple blocks world. More recent approaches have attempted to apply machine learning techniques to handle more complex language and target environments. For example, Lau et al. [40] use a multi-class classifier trained on manually annotated data to translate instructions into user-interface actions on a web page. MacMahon et al. [46] use supervised learning techniques to interpret route directions into navigation actions. More recently, Tellex et al. [71] leverage unaligned parallel data to learn a mapping from textual instructions to the actions of a physical robot. While such machine learning techniques are more robust than rule-based methods, these approaches are limited in their scalability due to their reliance on expensive human supervision in the form of rules or annotations.

In contrast to such prior work, our method learns directly from environment feedback as its only source of supervision. In fact, we show that in our domains, learning from environment feedback can yield a performance on par with an equivalent method trained on manual annotations. We compute this environment feedback signal via a simple heuristic which looks at text overlap between the instruction text and text labels in the target environment. The assumption that the instruction text corresponds to the target environment is the only prior knowledge our algorithm has about either the language or the environment. This allows our method to be applied to domains where a reasonable environment feedback can be automatically computed. Further-

more, our algorithm can also efficiently leverage manual annotations when available, effectively combining this supervision with the environment feedback.

2.2.3 Reinforcement Learning

Application to Natural Language Processing Reinforcement learning has previously been applied to the problem of dialogue management in natural language processing [59, 55, 45, 65]. These systems manage conversations with a human user in a step-by-step fashion, by selecting the computer's next natural language utterance. In this task, the reinforcement learning state space encodes information about the goals of the user, and the utterances of both the user and the computer up to the current time step. The RL action space is defined by a prespecified set of utterances available to the computer. The task in dialogue management is to find a policy that maps these RL states to actions and optimally achieves some predefined dialogue goal. The corresponding learning problem is to find this optimal policy through a trial-and-error process of repeated spoken interaction with the human user.

The application of reinforcement learning is different in several ways between dialogue systems and our setup. In some respects, our task is more easily amenable to reinforcement learning – for instance, we are not interacting with a human user, so the cost of interaction is lower. However, the structure of the RL state space makes our task fundamentally more challenging. While the state space can be designed to be relatively small in the dialogue management task, our state space is determined by the underlying environment and is typically very large. We address this complexity by developing a policy gradient algorithm that learns efficiently while exploring a small but relevant subset of the states.

Modeling the Target Environment Our work combines ideas of two traditionally disparate approaches to reinforcement learning [68]. The first approach, *model-based reinforcement learning*, constructs a model of the environment in which the learner operates (e.g., modeling location, velocity, and acceleration in robot navigation). It then computes a policy directly from the rich information represented in

the induced environment model. In the NLP literature, this approach is commonly used for dialog management [64, 42, 58]. Model-based learning is very effective when the environment can be efficiently estimated, and represented in a compact fashion. However, when this is not possible, due to difficulties in either estimation or representation, model-based methods perform poorly [9, 37]. Our instruction interpretation task falls into this latter category,[2] rendering standard model-based learning ineffective.

The second approach – i.e., *model-free reinforcement learning* methods such as policy learning – aims to select the optimal action at every step, without explicitly constructing a model of the environment. While policy learners can effectively operate in complex environments, they are not designed to benefit from a learnt environment model. We address this limitation by expanding a policy learning algorithm to take advantage of a partial environment model estimated during learning. Our approach of conditioning the policy function on future reachable states is similar in concept to the use of *post-decision state* information in the approximate dynamic programming framework [53]. This ability to explicitly condition action selection on the future consequences of actions allows our method to effectively interpret high-level instructions, while also improving performance on low-level instructions.

[2]For example, in the Windows GUI domain, clicking on the *File* menu will result in a different submenu depending on the application. Thus it is very difficult to predict the effects of a previously unseen GUI command.

2.3 Model

In this section, we first formally define our task in terms of the *Markov Decision Process* framework, and show how language analysis and environment command selection can be modeled jointly in this framework. Thereafter, we describe a model for interpreting low-level instructions, and detail its extension to high-level instructions.

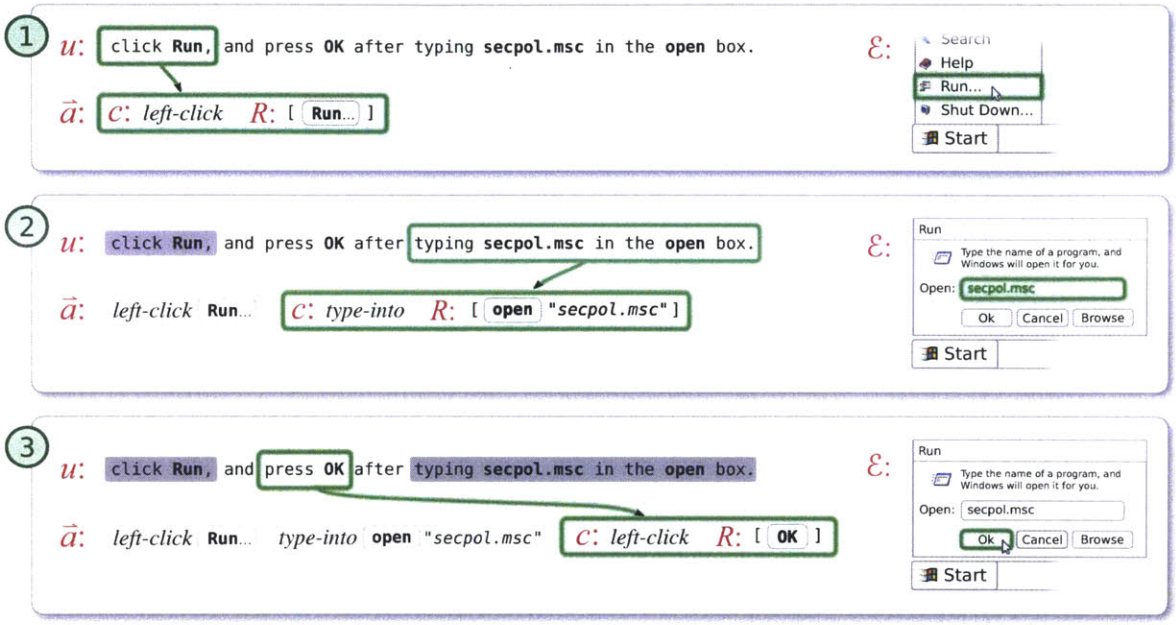
2.3.1 Problem Formulation

The input to our mapping task is a document d composed of sentences $(u_1, \ldots, u_\ell)$, where each u_i is a sequence of words. Our goal is to map d to a sequence of commands $\vec{c} = (c_0, \ldots, c_{n-1})$, which can be executed in a given target environment. We assume interactive access to this environment – i.e., that an algorithm can observe the environment's current state $\mathcal{E}$, and execute commands on it. We also assume knowledge of the environment's primitive commands, and the ability to identify the objects available for interaction given a state observation.

The correct interpretation of an instruction into a command can depend on the state $\mathcal{E}$ of the target environment in addition to the instruction text. Therefore, we represent the mapping task as a Markov Decision Process [3] (MDP) – a mathematical framework for sequential decision making under uncertainty [68]. Intuitively, we model the instruction mapping process as shown in Figure 2-2. At each step, conditioned on the text and the environment state, we first select the words which describe the next command to be executed, translate the selected words into the corresponding command, and execute the command in the environment.

Markov Decision Process for Instruction Interpretation Formally we define our MDP as the 4-tuple $\langle S, A, T, R \rangle$ where:

[3] In general, the MDP assumption of *full state observability* may not hold for all tasks in domains such as a graphical user interface – for example tasks which change internal system states. Such tasks are better modeled as Partially Observable Markov Decision Processes (POMDP). In this work, we limit our attention to tasks which fit into the MDP formulation, leaving POMDP formulations of instruction interpretation for future work.



$d = (u_1, \ldots, u_\ell)$	: Document	c	: Command
u	: Sentence	R	: Command parameters
$\mathcal{E}$	: Environment state	W_c	: Words mapped to action
$\vec{a} = (a_0, \ldots, a_{n-1})$	: Sequence of actions	W	: Words mapped to previous actions
$a = (c, R, W_c)$	: Action		

Figure 2-2: An example of a sentence containing three low-level instructions being mapped to a sequence of actions in Windows 2000. For each step, the figure shows the words selected by the action, along with the corresponding system command and its parameters. The words of W_c are highlighted by green rectangles with an arrow pointing to the corresponding command, and the words of W are highlighted in grey.

- ***State space***, S, is the set of all possible states. Each state $s \in S$ represents the information used to select the next environment command c_i. As such, this *mapping state* s needs to encompass both the instruction text as well as the state of the target environment. We therefore define s as the tuple $(\mathcal{E}, d, j, W)$, where $\mathcal{E}$ refers to the current environment state; j is the index of the sentence currently being interpreted in document d; and W is the set of words which have already been mapped. W allows us to keep track of the interpretation process, and stops the same words from being mapped to multiple commands. The mapping state s is observed before each command selection step.

- ***Action space***, A, is the set of all possible actions. A *mapping action* a is the tuple (W_c, c, R), where W_c are the words describing the next command to execute, c is the corresponding environment command, and R are its parameters. Elements of R refer to objects present in environment state $\mathcal{E}$, and also possibly words in document d. For example, in the second step of Figure 2-2, R refers to a GUI *textbox* named "open", as well as the text "secpol.msc" which needs to be typed into the textbox. To account for words that do not describe any commands, c can also take the special value *null*.

- ***Transition distribution***, $T(s' \mid s, a)$ encodes the way the mapping state $s = (\mathcal{E}, d, j, W)$ changes to a new state s' in response to action $a = (W_c, c, R)$. We define this distribution as follows: W is updated with a's selected words W_c; j is incremented if all words of the sentence have been mapped; and $\mathcal{E}$ changes according to the environment's transition distribution $p(\mathcal{E}'|\mathcal{E}, c, R)$. This transition distribution is not known *a priori*, but can be observed by interacting with the environment. As we will see in Section 2.3.4, our approach avoids having to directly estimate this distribution. For the applications we consider in this work, environment state transitions, and consequently mapping state transitions, are deterministic.

- ***Reward function***, $R(h) \in \mathbb{R}$, defines the feedback or *reward* received after completing the interpretation of a document d. Here, $h = (s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n)$

is a *history* of states and actions visited during the mapping process.[4] The value of the reward correlates with the goodness of the mapping, with higher rewards indicating better mappings. This reward function will serve as the supervision signal from which we learn instruction mapping. Section 2.4 describes how effective reward functions can be computed for our test domains using simple heuristics. We will also demonstrate how manually annotated action sequences can be incorporated into the reward.

Under the MDP representation of instruction interpretation, the goal is to estimate the parameters θ of an action selection distribution or *policy* $p(a \mid s; \theta)$. Here s is the current mapping state of the MDP, a is the mapping action to be taken under s, and policy $p(a \mid s; \theta)$ encodes the probability that executing a under s will lead to the correct interpretation of the given document.

2.3.2 A Policy for Interpreting Low-level Instructions

Our definition above of the state and action spaces of the MDP results in a policy structure designed to address the three challenges of instruction interpretation – segmentation, translation and reordering. In particular, given the definitions of state $s = (\mathcal{E}, d, j, W)$ and action $a = (W_c, c, R)$, we have:

$$\begin{aligned} p(a \mid s; \theta) &= p(W_c, c, R \mid \mathcal{E}, d, j, W; \theta) \\ &= p(W_c \mid \mathcal{E}, d, j, W; \theta)\ p(c, R \mid W_c, \mathcal{E}; \theta). \end{aligned} \tag{2.1}$$

Here we assume that given the words W_c, the command c and parameters R are conditionally independent of the document text – i.e., document d and sentence j. The first component of this policy selects the segment of words W_c to be mapped in the

[4] In most reinforcement learning problems, the reward function is defined over state-action pairs, as $r(s, a)$ — in this case, $r(h) = \sum_t r(s_t, a_t)$, and our formulation becomes a standard finite-horizon Markov Decision Process. Policy gradient approaches allow us to learn using the more general case of history-based reward.

current interpretation step. This segment can be selected from any part of the current sentence j, implicitly allowing segmentation and reordering. Selection of W_c is conditioned on the text, the words that have already been mapped, and on the state of the environment, making it sensitive to both the text and environment context. The second component in Equation 2.1 performs the translation step of instruction mapping – i.e., by selecting an environment command c and the corresponding parameters R given words W_c and the environment state $\mathcal{E}$.

We represent the policy $p(a \mid s; \theta)$ in a log-linear fashion [23, 39], giving us the flexibility to leverage a diverse range of features to model action selection. Under this representation, the policy distribution is defined by:

$$p(a \mid s; \theta) = \frac{e^{\theta \cdot \vec{\phi}(s,a)}}{\sum_{a'} e^{\theta \cdot \vec{\phi}(s,a')}}, \tag{2.2}$$

where $\vec{\phi}(s, a) \in \mathbb{R}^n$ is an n-dimensional feature function. Given parameters θ, and a mapping state $s = (\mathcal{E}, d, j, W)$, this policy distribution can be computed by enumerating out the space of possible actions $a = (W_c, c, R)$ under s. This action space is defined by the Cartesian product of all subspans W_c of unused words in the current sentence (i.e., subspans of the j^{th} sentence of d not in W), and the possible commands c and parameters R in environment state $\mathcal{E}$.[5] The details of how this action and state space are defined for specific instruction interpretation tasks are given in Section 2.4.

Given the parameters of the policy, we can map a document into a command sequence by repeatedly choosing an action a given the current mapping state s, and applying that action to advance to a new state s' (See Figure 2-2). This processes is continued until we reach a state from which no further mappings are possible. As described in Section 2.4, such *dead-end* states can be easily, albeit approximately, identified in our target domains. During testing, actions are selected according to the mode of the policy distribution. However, as described in the following section,

[5]For parameters that refer to words, the space of possible values is defined by the unused words in the current sentence.

during learning a certain amount of randomness is introduced into the action selection process to encourage exploration of previously untried actions.

2.3.3 Extending the Policy to High-level Instructions

The main challenge in processing high-level instructions is that in contrast to their low-level counterparts, they correspond to sequences of one or more commands. A simple way to enable this one-to-many mapping is to allow commands that do not consume words (i.e., $|W_c| = 0$). The sequence of commands can then be constructed incrementally using the policy described in the previous section (Section 2.3.2). However, this change significantly complicates the interpretation problem — we need to be able to predict commands that are not directly described by any words, and allowing such action sequences significantly increases the space of possibilities for each instruction. Since we cannot enumerate all possible sequences at decision time, we limit the space of possibilities by learning which sequences are likely to be relevant for the current instruction.

To motivate the approach, consider the decision problem in Figure 2-3, where we need to find a command sequence for the high-level instruction "open control panel." Our algorithm narrows down the search space by focusing on sequences that lead to environment states where the control panel icon was previously observed. Information about such states is acquired during learning in the form of a *partial environment model* $q(\mathcal{E}' \mid \mathcal{E}, c)$.

Our goal is to map high-level instructions to command sequences by leveraging knowledge about the long-term effects of commands. We do this by integrating the partial environment model into the policy function. Specifically, we modify the log-linear policy from Equation 2.1 to $p(a \mid s; q, \theta)$ by adding *look-ahead features* $\phi(s, a, q)$ which complement the local features used in the previous model. These look-ahead features incorporate various measurements that characterize the potential of future states reachable via the selected action. Although primarily designed to analyze high-level instructions, this approach is also useful for mapping low-level instructions.

Below, we first describe how we estimate the partial environment transition model

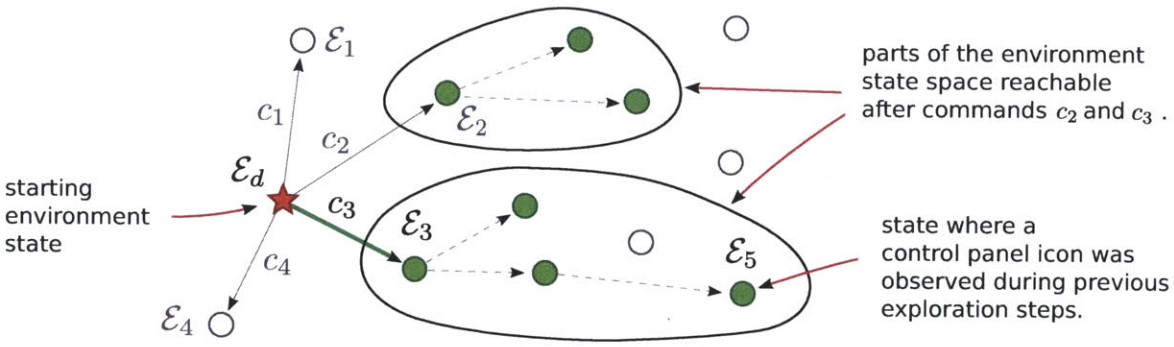


Figure 2-3: Using information derived from future states to interpret the high-level instruction "open control panel." $\mathcal{E}_d$ is the starting state, and c_1 through c_4 are candidate commands. Environment states are shown as circles, with previously visited environment states colored green. Dotted arrows show known state transitions. All else being equal, the information that the control panel icon was observed in state $\mathcal{E}_5$ during previous exploration steps can help to correctly select command c_3.

and how this model is used to compute the look-ahead features. This is followed by the details of parameter estimation for our algorithm.

Partial Environment Transition Model

To compute the look-ahead features, we first need to collect statistics about the environment transition function $p(\mathcal{E}' \mid \mathcal{E}, c)$. An example of an environment transition is the change caused by clicking on the "start" button. We collect this information through observation, and build a partial environment transition model $q(\mathcal{E}' \mid \mathcal{E}, c)$.

One possible strategy for constructing q is to observe the effects of executing random commands in the environment. In a complex environment, however, such a strategy is unlikely to produce state samples relevant to our text analysis task. Instead, we use the training documents to guide the sampling process. During training, we execute the command sequences predicted by the policy function in the environment, caching the resulting state transitions. Initially, these commands may have little connection to the actual instructions. As learning progresses and the quality of the interpretation improves, more promising parts of the environment will be observed.

This process yields samples that are biased toward the content of the documents.

Look-Ahead Features

We wish to select actions that allow for the best follow-up actions, thereby finding the analysis with the highest total reward for a given document. In practice, however, we do not have information about the effects of all possible future actions. Instead, we capitalize on the state transitions observed during the sampling process described above, allowing us to incrementally build an environment model of actions and their effects.

Based on this transition information, we can estimate the usefulness of actions by considering the properties of states they can reach. For instance, some states might have very low immediate reward, indicating that they are unlikely to be part of the best analysis for the document. While the usefulness of most states is hard to determine, it correlates with various properties of the state. We encode the following properties as look-ahead features in our policy:

- The highest reward achievable by an action sequence passing through this state. This property is computed using the learnt environment model, and is therefore an approximation.

- The length of the above action sequence.

- The average reward received at the environment state while interpreting any document. This property introduces a bias towards commonly visited states that frequently recur throughout multiple documents' correct interpretations.

Because we can never encounter all states and all actions, our environment model is always incomplete and these properties can only be computed based on partial information. Moreover, the predictive strength of the properties is not known in advance. Therefore we incorporate them as separate features in the model, and allow the learning process to estimate their weights. In particular, we select actions a based

on the current state s and the partial environment model q, resulting in the following policy definition:

$$p(a \mid s; q, \theta) = \frac{e^{\theta \cdot \phi(s,a,q)}}{\sum_{a'} e^{\theta \cdot \phi(s,a',q)}},$$

where the feature representation $\phi(s, a, q)$ has been extended to be a function of q. We factor this policy function into a product of experts [35] as follows:

$$p(a \mid s, q; \theta) = p(a \mid s; \theta)\ p_l(a \mid s, q; \theta). \tag{2.3}$$

Here, the first term models action selection when the environment command is directly specified by the text. This term is identical to our policy for mapping low-level instructions from Equation 2.1. The second term, $p_l(a \mid s, q; \theta)$, allows the algorithm to select actions based on the partial environment model q when textual information is insufficient.

2.3.4 Parameter Estimation via Reinforcement Learning

During training the algorithm is provided with a set of documents $d \in D$, an environment in which to execute command sequences $\vec{c}$, and a reward function $r(h)$. Our goal is to estimate two sets of parameters: 1) the optimal parameters θ^* of the policy function, and 2) the partial environment transition model $q(\mathcal{E}' \mid \mathcal{E}, c)$. Since the latter is defined as the observed portion of the true deterministic model $p(\mathcal{E}' \mid \mathcal{E}, c)$, estimating it is straightforward – we can simply memorize all environment transitions observed during our interactions with the world.

To estimate θ^*, we observe that these are defined as the parameters of the policy most likely to interpret a given document correctly. Since the reward $r(h)$ observed after mapping a document correlates with the correctness of the mapping, a natural objective during learning is to maximize the expected future reward or *Value function.* Formally, we define the Value function $V_\theta(s_0)$ as the reward we expect to receive while

acting according to a given policy starting from state s_0:

$$V_\theta(s_0) \;=\; \mathbb{E}_{p(h|\theta)}\left[r(h)\right]. \tag{2.4}$$

Here history $h = \{s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n\}$ is the sequence of states and actions encountered while interpreting a single document, starting from s_0. The distribution $p(h \mid \theta)$ is the probability of seeing history h when starting from state s_0 and acting according to a policy with parameters θ. This distribution can be decomposed into a product over time steps:

$$p(h \mid \theta) \;=\; \prod_{t=0}^{n-1} p(a_t \mid s_t; \theta)\;\; p(s_{t+1} \mid s_t, a_t), \tag{2.5}$$

$$\;=\; \prod_{t=0}^{n-1} p(a_t \mid s_t; \theta)\;\; T(s_{t+1} \mid s_t, a_t), \tag{2.6}$$

where the first term is the policy, and the second term is in fact the transition distribution $T(s' \mid s, a)$ of the MDP.

Given the above definition of the value function, our learning objective is to maximize the total expected future reward over all documents – i.e., the optimal policy parameters are given by

$$\theta^* \;=\; \arg\max_\theta \sum_{d \in D} V_\theta(s_d), \tag{2.7}$$

where $s_d = (\mathcal{E}_d, d, 0, \emptyset)$ is the special starting state of the MDP for document d, and $\mathcal{E}_d$ is the corresponding starting state of the target environment. This objective is dependent on the transition function $T(s_{t+1} \mid s_t, a_t)$ of the MDP. Since $T(s_{t+1} \mid s_t, a_t)$ is not known a priori, the objective function cannot be computed in closed form. However, the optimal parameters, θ^*, can be estimated approximately via several well studied reinforcement learning techniques. *Policy gradient* algorithms constitute one class of such techniques.

A Policy Gradient Algorithm for Instruction Interpretation

Policy gradient algorithms are a class of efficient reinforcement learning methods for approximately estimating the optimal policy parameters θ. These algorithms estimate θ by performing stochastic gradient ascent on the value function V_θ. They approximate the gradient of V_θ by interacting with the target environment and observing the resulting reward signal. Policy gradient algorithms optimize a non-convex objective and are only guaranteed to find a local optimum. However, they scale to large state spaces and as we will see, in practice, they perform well in the instruction interpretation task.

To find the parameters θ that maximize the objective, we first compute the derivative of V_θ (see Appendix A.1 for details). Expanding according to the product rule, we have:

$$\frac{\partial}{\partial\theta}V_\theta(s) \;=\; \mathbb{E}_{p(h|\theta)}\left[r(h)\sum_t \frac{\partial}{\partial\theta}\log p(a_t \mid s_t;\theta)\right], \tag{2.8}$$

where the inner sum is over all time steps t in the current history h. Expanding the inner partial derivative we observe that:

$$\frac{\partial}{\partial\theta}\log p(a \mid s;\theta) \;=\; \vec{\phi}(s,a) - \sum_{a'}\vec{\phi}(s,a')p(a' \mid s;\theta), \tag{2.9}$$

which is the derivative of a log-linear distribution.

Equation 2.9 is easy to compute directly. However, the complete derivative of V_θ in Equation 2.8 is intractable, because computing the expectation would require summing over all possible histories. Instead, policy gradient algorithms employ stochastic gradient ascent by computing a noisy estimate of the expectation using just a subset of the histories. Specifically, we draw samples from $p(h|\theta)$ by acting in the target environment, and use these samples to approximate the expectation in equation 2.8. In practice, it is often sufficient to sample a single history h for this approximation. Under this approximation, using a learning rate of α, the parameter update equation

becomes

$$\Delta\theta \;=\; \alpha\, r(h) \sum_{t=0}^{n-1} \left(\phi_k(a_t, s_t) - \sum_{a \in A} \phi_k(a, s_t)\; p(a \mid s_t, \theta) \right). \tag{2.10}$$

Algorithm 1 shows the procedure for joint learning of these parameters. As in standard policy gradient learning [69], the algorithm iterates over all documents $d \in D$ (steps 1, 2), selecting and executing actions in the environment (steps 3 to 6). The resulting reward is used to compute the empirical gradient and update the parameters θ (steps 8, 9). The environment interactions also yields samples of state transitions which are used to estimate the partial environment model $q(\mathcal{E}' \mid \mathcal{E}, c)$ (step 7). This updated q is then used to compute the feature functions $\phi(s, a, q)$ during the next iteration of learning (step 4). This process is repeated until the total reward on the training documents converges. This algorithm capitalizes on the synergy between θ and q. As learning proceeds, the method discovers a more complete state transition function q, which improves the accuracy of the look-ahead features, and ultimately, the quality of the resulting policy. An improved policy function in turn produces state samples that are more relevant to the document interpretation task.

Efficient use of Environment Feedback

Each of the history samples used for the parameter updates requires our algorithm to actually interact with the target environment. In many domains, including the ones we test on, this interaction can be expensive. We therefore employ two techniques to take maximum advantage of each interaction. First, a history $h = (s_0, a_0, \ldots, s_n)$ contains subsequences $(s_i, a_i, \ldots, s_n)$ for $i = 1$ to $n - 1$, each with its own reward value given by the environment as a side effect of executing h. We apply the update from equation 2.9 for each such subsequence. Second, under our model, a single environment command sequence can be produced via different word mappings W_c of a given text. Thus, for a sampled history h, we can propose alternative histories h' that result in the same commands c and parameters R with different word spans W_c. These alternative histories will have a probability of occurrence $p(h' \mid \theta)$ that is

Input: A document set D,

Feature function $\vec{\phi}$,

Reward function $r(h)$,

Number of iterations N

Learning rate α

Initialization: Set θ to small random values.

Set q to the empty set.

1 **for** $i = 1 \cdots N$ **do**

2 **foreach** $d \in D$ **do**

Sample history $h \sim p(h|\theta)$ where

$h = (s_0, a_0, \cdots, a_{n-1}, s_n)$ as follows:

Initialize environment to document specific starting state $\mathcal{E}_d$

3 **for** $t = 0 \cdots n-1$ **do**

4 Compute $\phi(a, s_t, q)$ based on latest q

5 Sample action $a_t \sim p(a|s_t; q, \theta)$

6 Execute a_t on state s_t: $s_{t+1} \sim p(s|s_t, a_t)$

7 Set $q = q \cup \{(\mathcal{E}', \mathcal{E}, c)\}$ where $\mathcal{E}'$, $\mathcal{E}$, c are the environment states and commands from s_{t+1}, s_t, and a_t

end

8 $$\Delta\theta \leftarrow \alpha\, r(h) \sum_t \left[\vec{\phi}(s_t, a_t, q) - \sum_{a'} \vec{\phi}(s_t, a', q)\; p(a'|s_t; q, \theta) \right]$$

9 $\theta \leftarrow \theta + \Delta\theta$

end

end

Output: Estimate of parameters θ

Algorithm 1: A policy gradient algorithm for estimating the parameters of our model.

different from the original sampled history h. As we will see in the following section, parameter updates to our model depend on histories h being sampled according to their probability under the current policy. Therefore, we apply equation 2.9 to each h', weighted by its probability under the current policy, $\frac{p(h'|\theta)}{p(h|\theta)}$.

2.3.5 Reward Functions and ML Estimation

A particular advantage of our algorithm is its ability to effectively learn from a mix of both environment reward and manual annotations. At one extreme, our model is able to learn solely based on environment reward, which is our primary focus in this work. At the other extreme, when all documents are fully annotated, learning in our model is equivalent to stochastic gradient ascent with a maximum likelihood objective. As shown by our results (see Figure 2-8), this ability to learn from mixed supervision allows our method to surpass the performance of an equivalent algorithm that learns only from manual annotations. We describe below the connection between ML estimation and our method when learning from annotated data.

Our model is able to learn from a range of reward functions, depending on the availability of annotated data and environment feedback. Consider the case when every training document $d \in D$ is annotated with its correct sequence of actions, and state transitions are deterministic. Given such annotations, it is straightforward to construct a reward function that connects policy gradient to maximum likelihood. Specifically, define a reward function $r(h)$ that returns one when h matches the annotation for the document being analyzed, and zero otherwise:

$$r(h) = \begin{cases} 1 & \text{if } h = h_d \\ 0 & \text{otherwise} \end{cases}.$$

where h_d is the history corresponding to the annotated action sequence. Policy gradient performs stochastic gradient ascent on the objective from equation 2.4, performing

one update per document. For document d, this objective becomes:

$$\begin{aligned} E_{p(h \mid \theta)}[r(h)] &= \sum_h r(h)\, p(h \mid \theta), \\ &= p(h_d \mid \theta), \end{aligned}$$

Thus, with this reward, policy gradient is equivalent to stochastic gradient ascent with a maximum likelihood objective.

2.3.6 Alternative Modeling Options

There are several alternative approaches to modeling the task of instruction interpretation. Here we discuss two such methods which we considered, but did not employ due to the characteristics of the task as mentioned below.

Global Inference Over Text Our models described in Sections 2.3.2 and 2.3.3, while leveraging global information from text via features, make greedy local interpretation decisions. Since such local decisions need not always be optimal, an obvious alternative is to perform global inference over the interpretation decisions. This, however, is complicated by the fact that the text interpretation decisions need to be conditioned on the current environment state, which changes with the execution of each interpreted command. Thus a model of the target environment is required for global inference over text. While this environment model need not be complete, it has to contain all the information relevant to the documents being interpreted. Automatically acquiring such an environment model is particularly difficult due to the large state space, and because we do not know the required portions of the environment prior to interpreting the documents.

Action-Value Function Approximation An alternative to our policy gradient approach to instruction interpretation is to learn an approximate action-value function – i.e., learning a approximation $Q'(s, a)$ of the action-value function $Q^{\pi}(s, a)$. Since the domain of $Q'(s, a)$ is the combination of the state and action spaces, the

approximation has to have sufficient capacity to represent the characteristics of the entire environment dynamics. In a complex environment such as the Windows operating system, the functionality of different parts of the system are by nature distinct. Thus constructing an approximation $Q'(s, a)$, that is compact and generalizes well across states is inherently difficult. In contrast, the domain of the corresponding policy function $\pi(a, s) = p(a \mid s)$ is only the action space. This makes a policy function approximation, as used by our methods, significantly easier to learn.

2.4 Applying the Model

We study two applications of our model – following instructions to perform software tasks, and solving a puzzle game using tutorial guides.

2.4.1 Microsoft Windows Help Domain

On its Help and Support website,[6] Microsoft publishes a number of articles describing how to perform tasks and troubleshoot problems in the Windows operating systems. Examples of such tasks include installing patches and changing security settings. Figure 2-4 shows one such article. Our goal is to automatically execute support articles for the Windows 2000 operating system on a machine running that OS.

Environment States and Commands We define the environment state $\mathcal{E}$ as the set of objects visible in the graphical user interface (GUI), along with the objects' properties such as label, location, and parent window. This information can be retrieved programmatically via standard OS APIs. The set of possible commands in this domain are *left-click*, *right-click*, *double-click*, and *type-into*, all of which take a GUI object as a parameter, while *type-into* additionally requires a parameter for the input text.

Policy Given the above environment state and action spaces, we can rewrite the MDP action definition as follows:

$$\begin{aligned} a &= (W_c, c, R) \\ &= (\{w_c, w_o, w_p\}, c, \{o, w_p\}), \\ \text{with } W_c &= \{w_c, w_o, w_p\}, \\ \text{and } R &= \{o, w_p\}. \end{aligned}$$

[6] support.microsoft.com

- Right click "My Computer" on the desktop, and click the Manage menu option.
- Click Services after expanding "Services and Applications".
- Set the remote registry service to start automatically and then start the service.

Figure 2-4: A Windows troubleshooting article describing how to configure the "remote registry service" to start automatically.

Here o is the GUI object to which command c should be applied; w_c and w_o are the words that describe c and o respectively; and w_p is the parameter text if $c = \textit{type-into}$ and $w_p = \textit{null}$ otherwise. We assume that command c and object o are conditionally independent of each other given the words that describe them, i.e., w_c and w_o. We also assume that the parameter text w_p is conditionally independent of object o given the object word w_o. These assumptions allow us to factor our policy function from Equation 2.3,

$$p(a \mid s, q; \theta) \;=\; p(a \mid s; \theta)\;\; p_l(a \mid s, q; \theta),$$

in the following fashion:

$$\begin{aligned} p(a \mid s; \theta) \;&=\; p(w_o, w_c \mid s; \theta) \times \\ &\qquad p(o \mid w_o, s; \theta)\; p(c \mid o, w_c, s; \theta) \times \\ &\qquad p(w_p \mid c, w_o, w_c, s; \theta), \\ p_l(a \mid s, q; \theta) \;&=\; p(c, o, w_p \mid w_c, w_o, W', q; \theta). \end{aligned}$$

We use the above factorization to efficiently compute the full policy, from which actions are subsequently selected.

Features In addition to the look-ahead features described in Section 2.3.3, we use several local features that capture various aspects of the action under consideration, the current Windows GUI state, and the input instructions. Table 2.1 shows some examples of these local features. For example, one lexical feature measures the sim-

Notation	
o	Parameter referring to an environment object
L	Set of object class names (e.g. "button")
V	Vocabulary
W	Unmapped words in current sentence

Features on words W, and object o
Test if o is visible in s
Test if o has input focus
Test if o is in the foreground
Test if o was previously interacted with
Test if o came into existence since last action
Min. edit distance between $w \in W$ and object labels in s
Features on given word w, command c, and object o
$\forall c' \in C$, $w' \in V$: test if $c' = c$ and $w' = w$
$\forall c' \in C, l \in L$: test if $c' = c$ and l is the class of o

Table 2.1: Example features in the Windows domain. All features are binary, except for the normalized edit distance which is real-valued.

ilarity of a word in the sentence to the GUI labels of objects in the environment. Environment-specific features, such as whether an object is currently in focus, are useful when selecting the object to manipulate. In total, there are 4,438 features.

Reward Function Environment feedback can be used as a reward function in this domain. An obvious reward would be task completion (e.g., whether the stated computer problem was fixed). Unfortunately, verifying task completion is a challenging systems issue in its own right.

Instead, we rely on a noisy method of checking whether execution can proceed from one sentence to the next: at least one word in each sentence has to correspond to an object in the environment.[7] For instance, in the sentence from Figure 2-2 the

[7]We assume that a word maps to an environment object if the edit distance between the word and the object's name is below a threshold value.

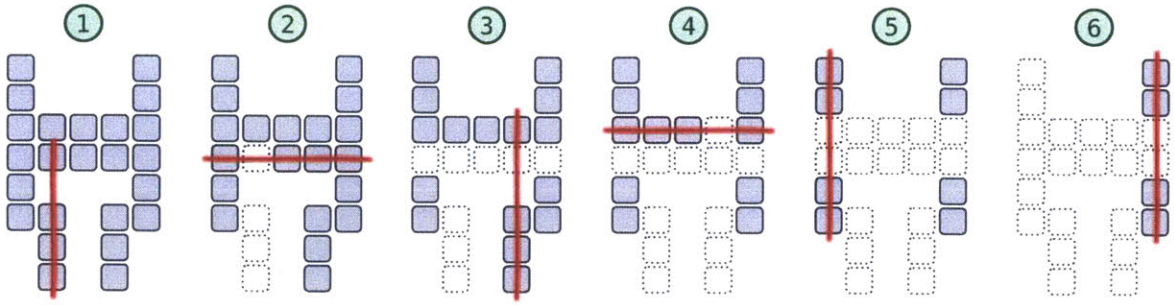
word "Run" matches the *Run...* menu item. If no words in a sentence match a current environment object, then one of the previous sentences is assumed to have been analyzed incorrectly. In this case, we assign the history a reward of -1.[8] When at least one word-match exists, we give the history a positive reward value. This value linearly increases with the percentage of words assigned to non-null commands, and linearly decreases with the number of output actions. This reward signal encodes the intuitions that most words in an instruction text will describe relevant environment commands, and that unnecessary commands will not be specified in the text. Although noisy, this reward signal is a powerful source of supervision for instruction mapping.

Formally we define the reward function as,

$$
\begin{aligned}
r(h) &= \sum_{i=1}^{\ell-1} r_d(u_i, u_{i+1}), \\
\text{where } r_d(u_i, u_{i+1}) &= \begin{cases} \frac{|W|}{|u_i|}\left(1 - \frac{n_c}{|u_i|}\right) & \text{if } u_{i+1} \text{ has word matches,} \\ -1 & \text{otherwise.} \end{cases}
\end{aligned} \tag{2.11}
$$

Here the document d being interpreted is composed of sentences $(u_1, \ldots, u_\ell)$; h is a candidate history; n_c is the number of commands executed in the environment for sentence u_i; and W is the set of words in the current history corresponding to those commands. Note that $\frac{|W|}{|u_i|}$ in Equation 2.11 increases with the number of words mapped, encouraging our algorithm to find parameters that map as many words as possible into commands. Similarly, $1 - \frac{n_c}{|u_i|}$ increases as fewer commands are executed per sentence, biasing our method away from executing unnecessary commands.

[8]This reward is not guaranteed to penalize all incorrect histories, because there may be false positive matches between the sentence and the environment.



Clear the bottom four from the second column from the left, the row of four, the second column from the right, the row of four, then the two columns.

Figure 2-5: An instance of the Crossblock puzzle showing its six step solution, and the text of the corresponding tutorial. For this level, four blocks in a row or column must be removed at once, and the goal is to remove all blocks. The red line shows the blocks to be removed at each step, and removed blocks are shown dotted.

2.4.2 Crossblock: A Puzzle Game Domain

Our second application is a puzzle game called *Crossblock*, available online as a Flash game.[9] Each of 50 puzzles is played on a grid, where some grid positions are filled with blocks. The object of the game is to clear the grid by drawing vertical or horizontal line segments that remove groups of blocks. Each segment must exactly cross a prespecified number of blocks, ranging from two to seven depending on the puzzle. Human players have found this game challenging and engaging enough to warrant posting textual tutorials.[10] An example of such a tutorial and its corresponding puzzle are shown in Figure 2-5. These tutorials, however, are composed entirely of low-level instructions. This domain is therefore used only to evaluate the performance of the algorithms on low-level instructions.

Environment States and Commands The environment is defined by the state of the game grid. The only command c is *clear*, which takes a parameter o specifying the grid location and orientation (*row* or *column*) of the line segment to be removed.

[9] http://hexaditidom.deviantart.com/art/Crossblock-108669149

[10] http://www.jayisgames.com/archives/2009/01/crossblock.php

The challenge in this domain is to segment the text into the phrases describing each action, and then correctly identify the line segments from references such as "the bottom four from the second column from the left."

Policy Given the structure of the environment states and commands described above, we rewrite the MDP action and policy as follows:

$$\begin{aligned} a &= (W_c, c, R) \\ &= (W_c, o), \\ \text{and } p(a \mid s; \theta) &= p(W_c \mid s; \theta)\, p(o \mid W_c, s; \theta). \end{aligned}$$

For this domain, we use two sets of binary features on state-action pairs (s, a). First, for each vocabulary word w, we define a feature that is one if w is the last word of a's consumed words W_c. These features help identify the proper text segmentation points between actions. Second, we introduce features for pairs of vocabulary word w and attributes of command parameter o, e.g., the line orientation and grid locations of the blocks that o would remove. This set of features enables us to match words (e.g., "row") with objects in the environment (e.g., a move that removes a horizontal series of blocks). In total, there are 8,094 features.

Reward Function For Crossblock it is easy to directly verify task completion, which we use as the basis of our reward function. The reward $r(h)$ is -1 if h ends in a state where the puzzle cannot be completed. For solved puzzles, the reward is a positive value proportional to the percentage of words assigned to non-null commands.

2.5 Experimental Setup

2.5.1 Datasets

For the Windows domain, our dataset consists of 188 documents, divided into 70 for training, 18 for development, and 100 for test. Of these test documents, 40 are composed solely of low-level instructions, while the remaining 60 contain at least one high-level instruction. In the puzzle game domain, we use 50 tutorials, divided into 40 for training and 10 for test.[11] Statistics for the datasets are shown below.

	Windows	Puzzle
Total # of documents	188	50
Total # of words	7448	994
Vocabulary size	739	46
Avg. words per sentence	9.93	19.88
Avg. sentences per document	4.38	1.00
Avg. actions per document	10.00	5.86

The data exhibits certain qualities that make for a challenging learning problem. For instance, a surprising variety of linguistic constructs are used to describe instructions in the text — as shown in Figure 2-6, even a simple command in the Windows domain is expressed in at least six different ways. Note that in the Crossblock domain, the entire sequence of instructions for each puzzle is written in a single sentence (see Figure 2-5 for an example), resulting in the sentences-per-document statistic of 1.00 shown above.

2.5.2 Reinforcement Learning Parameters

Following common practice, we encourage exploration during learning with an ϵ-greedy strategy [68], with ϵ set to 0.1. We also identify *dead-end* states, i.e. states

[11] For Crossblock, because the number of puzzles is limited, we did not hold out a separate development set, and report averaged results over five training/test splits.

```
On the tools menu, click internet options
Click tools, and then click internet options
Click tools, and then choose internet options
Click internet options on the tools menu
In internet explorer, click internet options on the tools menu
On the tools menu in internet explorer, click internet options
```

Figure 2-6: Variations of "click internet options on the tools menu" present in the Windows corpus.

with the lowest possible immediate reward, and use the induced environment model to encourage additional exploration by lowering the likelihood of actions that lead to such dead-end states.

During the early stages of learning, experience gathered in the environment model is extremely sparse, causing the look-ahead features to provide poor estimates. To speed convergence, we ignore these estimates by disabling the look-ahead features for a fixed number of initial training iterations.

Finally, to guarantee convergence, stochastic gradient ascent algorithms require a learning rate schedule. Thus we pick the learning rate using a modified *search-then-converge* algorithm [21], where we tie the learning rate to the ratio of training documents that received a positive reward in the current iteration.

2.5.3 Experimental Framework

To apply our algorithm to the Windows domain, we use the Win32 application programming interface to instrument the Windows 2000 user interface. This allows our learner to programmatically gather information about the environment state, and to execute mouse and keyboard commands. The operating system environment is hosted within a virtual machine,[12] allowing us to rapidly save and reset system state snapshots. Figure 2-7 shows the experimental framework for this domain. For the puzzle game domain, we replicated the game with an implementation that facilitates

[12]VMware Workstation, available at http://www.vmware.com

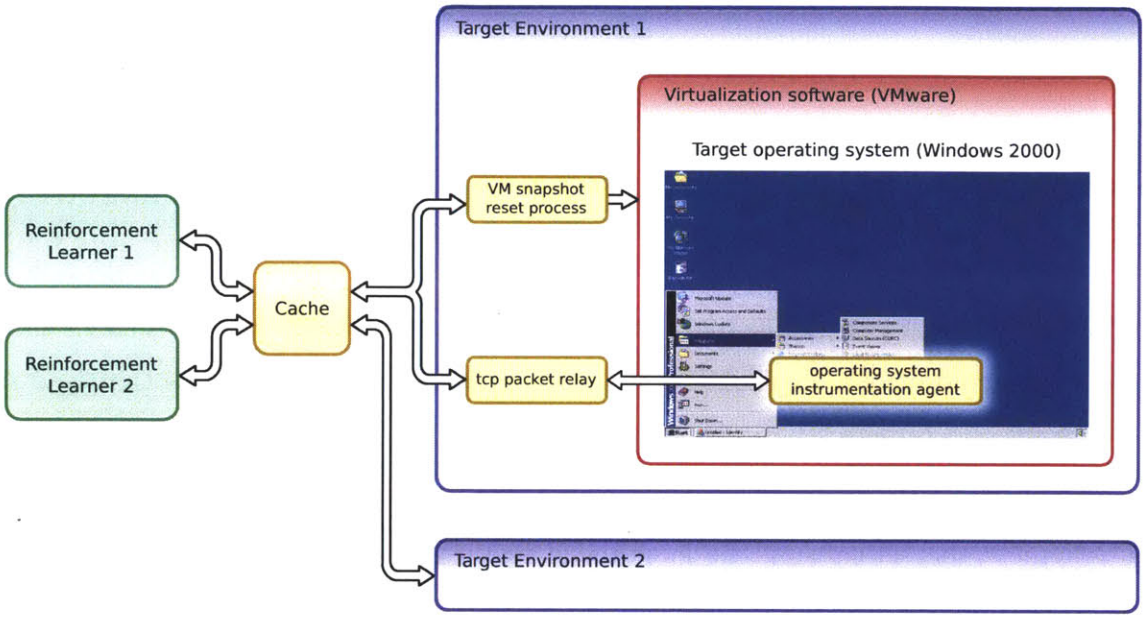


Figure 2-7: The framework used in the Windows 2000 experiments. The target environment, Windows 2000, is run within a virtual machine to allow the environment to be easily reset to its initial state. The *operating system instrumentation agent* allows the learner to observe the GUI state, and execute GUI commands. All information transfer between the different components of the framework are via TCP/IP. The reinforcement learner interacts with Windows 2000 via a *Cache*, speeds up the learning process by removing the need to interact with Windows 2000 for previously observed states and commands. The Cache also allows multiple learners to interact with multiple target environments in parallel, further reducing experiment run-times. The *tcp packet relay* is a simple message relay used to isolate the learner and cache from the effects of the virtual machine being repeatedly reset.

automatic play.

As is commonly done in reinforcement learning, we use a softmax temperature parameter to smooth the policy distribution [68], set to 0.1 in our experiments. For Windows, the development set is used to select the best parameters. For Crossblock, we choose the parameters that produce the highest reward during training. During evaluation, we use these parameters to predict mappings for the test documents.

2.5.4 Evaluation Metrics

For evaluation, we compare the results to manually constructed sequences of actions. We measure the number of correct actions, sentences, and documents. An action is correct if it matches the annotations in terms of commands and parameters. A sentence is correct if all of its actions are correctly identified, and analogously for documents.[13] In our instruction interpretation task, this evaluation is particularly onerous – each action depends on the correctness of all previous actions, so a single error can render the remainder of a document's mapping incorrect. Statistical significance is measured with the sign test.

Additionally, we compute a word alignment score to investigate the extent to which the input text is analyzed correctly. This score measures the percentage of words that are aligned to the corresponding annotated actions in correctly analyzed documents.

2.5.5 Baselines

We consider the following baselines to characterize the performance of our approach.

- **Full Supervision** Sequence prediction problems like ours are typically addressed using supervised techniques. We measure how a standard supervised approach would perform on this task by using a reward signal based on manual

[13] Due to variability in document lengths, overall action accuracy is not guaranteed to be higher than document accuracy. I.e., many short documents can be interpreted correctly by getting a few actions correct.

annotations of output action sequences, as defined in Section 2.3.5. As shown there, policy gradient with this reward is equivalent to stochastic gradient ascent with a maximum likelihood objective.

- **Partial Supervision** We consider the case when only a subset of the training documents is annotated, and the environment reward is used for the remainder. Our method seamlessly combines these two kinds of rewards.

- **Random and Majority (Windows)** We consider two naïve baselines. Both scan through each sentence from left to right. A command c is executed on the object whose name is encountered first in the sentence. If multiple objects match, one is selected at random. The command c to be executed is either selected *randomly*, or set to the *majority* command, which in the Windows 2000 domain is *left-click*. This procedure is repeated until no more words match environment objects.

- **Random (Puzzle)** We consider a baseline that *randomly* selects among the actions that are valid in the current game state. Since action selection is among objects, there is no natural majority baseline for the puzzle.

2.6 Results

To fully characterize our method, we evaluate several different aspects of its performance. We first analyze the accuracy with which our model interprets both high-level and low-level instructions. Given the importance of the environment model for interpreting high-level instructions, we then evaluate the impact of environment model quality on interpretation accuracy. Finally we investigate the underlying linguistic analysis performed by our method by evaluating the accuracy with which it selects the word spans of the commands.

2.6.1 Interpretation Performance

Table 2.2 presents evaluation results on the test sets, and for the sake of clarity, shows performance on low-level instructions and high-level instructions separately. There are several indicators of the difficulty of this task. The random and majority baselines' poor performance in both domains indicates that naïve approaches are inadequate for these tasks. The performance of the fully supervised approach provides further evidence that the task is challenging. This difficulty can be attributed in part to the large branching factor of possible actions at each step — on average, there are 27.14 choices per action in the Windows domain, and 9.78 in the Crossblock domain.

In both domains, the learners relying only on environment reward perform well – in the case of our full model, even rivaling the performance of the supervised equivalent. Particularly surprising is the comparison between environment-reward and manual supervision in the case of high-level instructions. As we will see from Section 2.6.3, the better performance of the environment-supervised method is due to the quality of the resulting partial environment model.

To characterize the environment reward signal in terms of the tradeoff between annotation effort and system performance, we perform an evaluation where only a portion of the documents are annotated. To avoid any confounding factors, we perform this experiment on the low-level instruction documents with the partial environment model switched off. Figure 2-8 shows the resulting tradeoff curve.

	Low-level instruction dataset			
	Windows		Puzzle	
	Action	Document	Action	Document
Random baseline	0.128	0.000	0.081	0.111
Majority baseline	0.287	0.100	—	—
No-env	$*$ 0.647	$*$ 0.375	$*$ 0.428	$*$ 0.453
No-env + annotation	$\diamond$ 0.756	0.525	0.632	0.630
Our model	0.793	0.517	—	—
Our model + annotation	0.793	0.650	—	—

	High-level instruction dataset		
	action	high-level action	document
Random baseline	0.000	0.000	0.000
Majority baseline	0.000	0.000	0.000
No-env	0.021	0.022	0.000
No-env + annotation	0.035	0.022	0.000
Our model	$*$ 0.419	$*$ 0.615	$*$ 0.283
Our model + annotation	$*$ 0.357	0.492	0.333

Table 2.2: Accuracy of the mapping produced by our model, its variants, and the baseline. Values marked with $*$ are statistically significant at $p < 0.01$ compared to the value immediately above it, while $\diamond$ indicates $p < 0.05$.

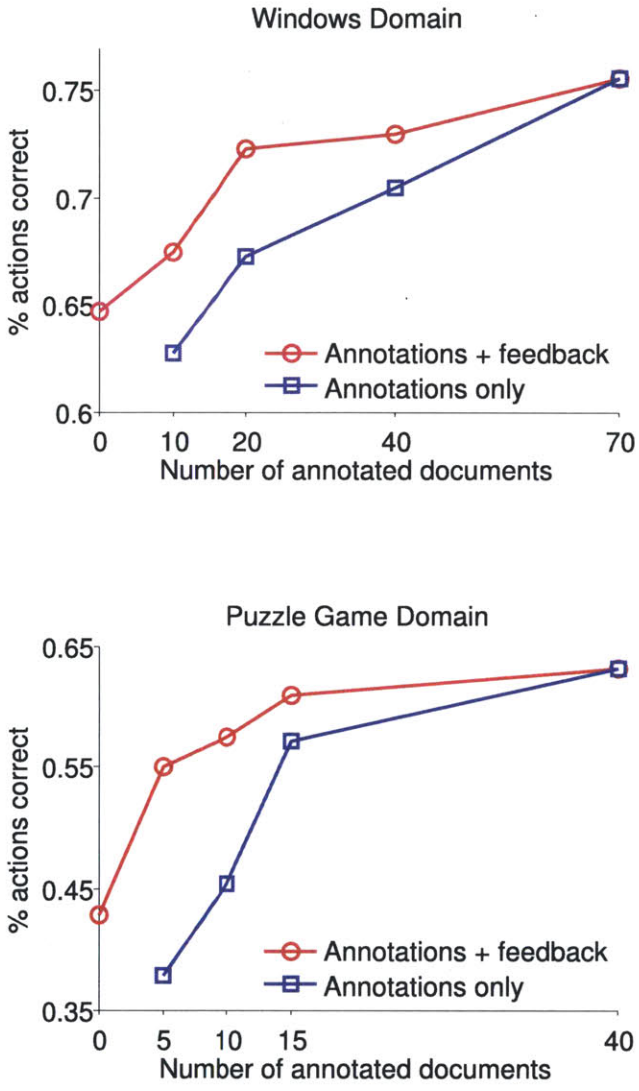


Figure 2-8: Comparison of two training scenarios where training is done using a subset of annotated documents, with and without environment reward for the remaining unannotated documents.

	Windows	Puzzle
Environment reward	0.819	0.686
Partial supervision	0.989	0.850
Full supervision	0.991	0.869

Table 2.3: The accuracy of our method's language analysis on the test set with different reward signals. While learning from manual annotations performs best, these results show the feasibility of learning language analysis based on noisy environment reward.

To further assess the contribution of the instruction text, we train a variant of our model without access to text features. This is possible in the game domain, where all of the puzzles share a single goal state that is independent of the instructions. This variant solves 34% of the puzzles, suggesting that access to the instructions significantly improves performance.

2.6.2 Accuracy of Linguistic Analysis

The word alignment results from Table 2.3 indicate that the learners are mapping the correct words to actions for documents that are successfully completed. For example, the models that perform best in the Windows domain achieve nearly perfect word alignment scores. While the variant of our method that learns from manual supervision performs best in this evaluation, the performance of the fully environment-supervised methods shows the feasibility of learning linguistic analysis based on such noisy supervision signals.

Finally, to demonstrate the quality of the learnt word–command alignments, we evaluate our method's ability to paraphrase from high-level instructions to low-level instructions. Here, the goal is to take each high-level instruction and construct a text description of the steps required to achieve it. We did this by finding high-level instructions where each of the commands they are associated with is also described by a low-level instruction in some other document. For example, if the text "open

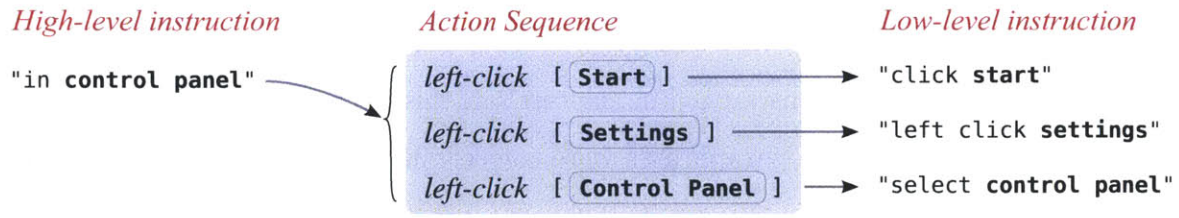


Figure 2-9: The process of paraphrasing a high-level instruction into a sequence of low-level instructions. After the high-level instruction has been mapped to executable commands, low-level descriptions of those commands from other documents are used to create the paraphrases.

High-level instruction
- open **device manager**

Extracted low-level instruction paraphrase
- double click **my computer**
- double click **control panel**
- double click **administrative tools**
- double click **computer management**
- double click **device manager**

High-level instruction
- open the **network tool** in **control panel**

Extracted low-level instruction paraphrase
- click **start**
- point to **settings**
- click **control panel**
- double click **network and dial-up connections**

Figure 2-10: Examples of automatically generated paraphrases for high-level instructions. The model maps the high-level instruction into a sequence of commands, and then translates them into the corresponding low-level instructions.

control panel" was mapped to the three commands in Figure 2-9, and each of those commands was described by a low-level instruction elsewhere, this procedure would create a paraphrase such as "click start, left click setting, and select control panel." Of the 60 high-level instructions tagged in the test set, this approach found paraphrases for 33 of them. 29 of these paraphrases were correct, in the sense that they describe all the necessary commands. Figure 2-10 shows some examples of the automatically extracted paraphrases.

2.6.3 Impact of Environment Model Quality

To validate the intuition that the partial environment model must contain information relevant for the language interpretation task, we replaced the learnt environment model with one of the same size gathered by executing random commands. The model with randomly sampled environment transitions performs poorly: it can process only 4.6% of the documents and 15.0% of the actions on the dataset with high-level instructions, compared to 28.3% and 41.9% respectively for our algorithm. This result also explains why training with full supervision hurts performance on high-level instructions (see Table 2.2). Learning directly from annotated command sequences results in a low-quality environment model due to the relative lack of exploration, hurting the model's ability to leverage the look-ahead features.

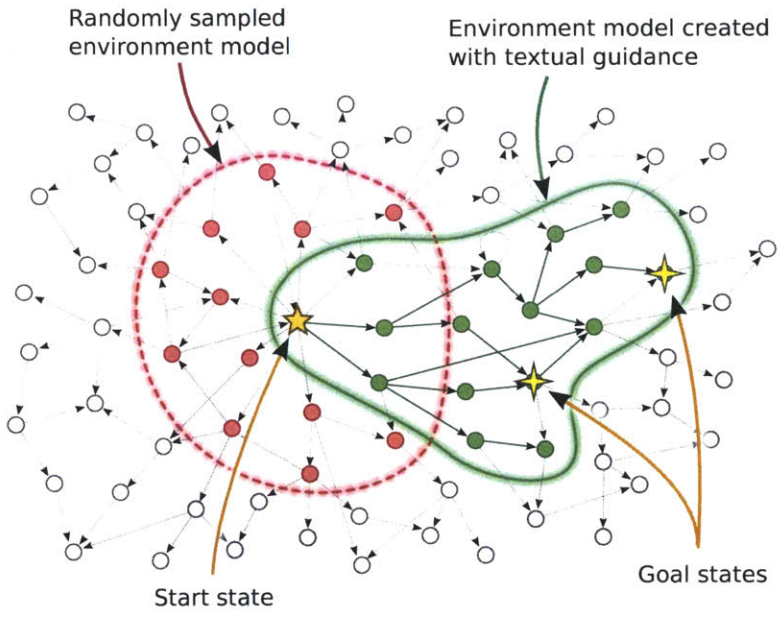


Figure 2-11: An illustration of the differences between an environment model constructed with textual guidance (shown in green), and one created via random exploration (shown in red). The nodes and edges of the graph symbolize the states and actions of the environment. Random exploration tends to construct a model of states evenly distributed around the starting state, which is denoted here by a star. In contrast, the model created with text guidance contains the neighborhood of states and actions described in the text. Since the textual instructions cover useful tasks, this neighborhood has a higher likelihood of being relevant to new unseen tasks.

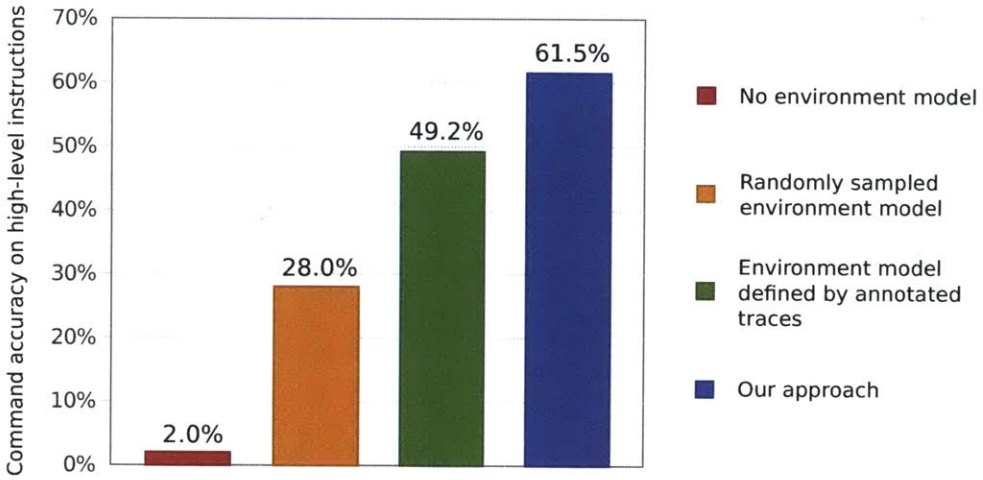


Figure 2-12: The performance of our method on high-level instructions when given various environment models.

2.7 Conclusion

In this chapter, we presented two reinforcement learning approaches for inducing a mapping between instructions and actions. Our methods are able to use environment-based rewards, such as task completion, to effectively learn text analysis. Our results show that having access to a suitable reward function can significantly reduce or even obviate the need for manual annotations. Furthermore, our results demonstrate the importance of modeling the grounding context (i.e., the target environment) when interpreting language that abstracts over low-level details. In addition to being effective at instruction interpretation, our method is also broadly applicable to domains where the correctness of the language grounding can be automatically evaluated based on environment feedback.

While our method is able to effectively learn only from environment feedback, the type of language it can handle is limited by two strong assumptions. First, we have assumed that the text contains imperative language, and explicitly describes a sequence of commands. Second we assume that all essential language grounding occurs at the object and command level – i.e., we aim to identify and map the descriptions of objects and commands from the text. These assumptions exclude a large fraction of text containing useful information about tasks in the world. For example, documents can contain non-imperative language, describing actions or behaviours that are generally useful in the world irrespective of any particular task. Moreover, the unit of grounding can be at the level of relationships between objects or actions. In the following two chapters of this thesis, we explore algorithms that can effectively learn language grounding in these more complex scenarios.

3

Interpreting Strategy Descriptions into Control Behaviour

In this chapter, we consider the task of automatically interpreting a strategy guide containing high-level situationally-relevant advice, and using this information to play a complex strategy game. Our Monte-Carlo Search method for textually-guided game play significantly outperforms strong text-unaware alternatives. We also show that while learning only from game feedback, our method is able to produce text analyses that conform to human notions of correctness.

3.1 Introduction

In this chapter, we study the task of grounding high-level situationally-relevant textual information in control applications such as computer games. In these applications, an agent attempts to optimize a utility function (e.g., game score) by learning to select situation-appropriate actions. In complex domains, finding a winning strategy is challenging even for humans. Therefore, human players typically rely on manuals and guides that describe promising tactics and provide general advice about the underlying task. Surprisingly, such textual information has never been utilized in control algorithms despite its potential to greatly improve performance. Our goal, therefore, is to develop methods that can achieve this in an automatic fashion. We explore this question in the context of strategy games, a challenging class of large

scale adversarial planning problems.

Consider for instance the text shown in Figure 3-1. This is an excerpt from the user manual of the game Civilization II.[1] This text describes game locations where the action *build-city* can be effectively applied. A stochastic player that does not have access to this text would have to gain this knowledge the hard way: it would repeatedly attempt this action in a myriad of states, thereby learning the characterization of promising state-action pairs based on observed game outcomes. In games with large state spaces, long planning horizons, and high-branching factors, this approach can be prohibitively slow and ineffective. An algorithm with access to the text, however, could learn correlations between words in the text and game attributes – e.g., the word "river" and places with rivers in the game – thus leveraging strategies described in text to select better actions.

To improve the performance of control applications using domain knowledge automatically extracted from text, we need to address the following challenges:

- **Grounding Text in the State-Action Space of a Control Application** Text guides provide a wealth of information about effective control strategies, including situation-specific advice as well as general background knowledge. To benefit from this information, an algorithm has to learn the mapping between the text of the guide, and the states and actions of the control application. This mapping allows the algorithm to find state-specific advice by matching state attributes to their verbal descriptions. Furthermore, once a relevant sentence is found, the mapping biases the algorithm to select the action proposed in the guide document. While this mapping can be modeled at the word-level, ideally we would also use information encoded in the structure of the sentence – such as the predicate argument structure. For instance, the algorithm can explicitly identify predicates and state attribute descriptions, and map them directly to structures inherent in the control application.

- **Annotation-free Parameter Estimation** While the above text analysis tasks

[1] http://en.wikipedia.org/wiki/Civilization_II

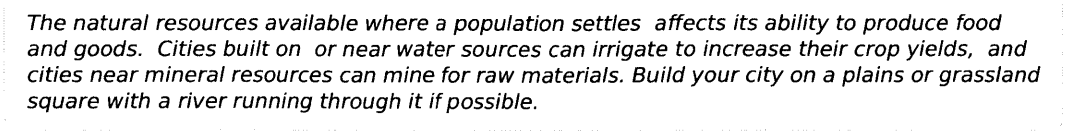


Figure 3-1: An excerpt from the user manual of the game Civilization II.

relate to well-known methods in information extraction, prior work has primarily focused on supervised methods. In our setup, text analysis is state dependent, therefore annotations need to be representative of the entire state space. Given an enormous state space that continually changes as the game progresses, collecting such annotations is impractical. Instead, we propose to learn text analysis based on a feedback signal inherent to the control application, e.g., the game score. This feedback is computed automatically at each step of the game, thereby allowing the algorithm to continuously adapt to the local, observed game context.

- **Effective Integration of Extracted Text Information into the Control Application** Most text guides do not provide complete, step-by-step advice for all situations that a player may encounter. Even when such advice is available, the learnt mapping may be noisy, resulting in suboptimal choices. Therefore, we need to design a method which can achieve effective control in the absence of textual advice, while robustly integrating automatically extracted information when available. We address this challenge by incorporating language analysis into *Monte-Carlo Search*, a state-of-the-art framework for playing complex games. Traditionally this framework operates only over state and action features. By extending Monte-Carlo search to include textual features, we integrate these two sources of information in a principled fashion.

Summary of Approach We address the above challenges in a unified framework based on *Markov Decision Processes* (MDP), a formulation commonly used for game playing algorithms. This setup consists of a game in a stochastic environment, where the goal of the player is to maximize a given utility function $R(s)$ at state s. The

player's behaviour is determined by an action-value function $Q(s, a)$ that assesses the goodness of action a at state s based on the attributes of s and a.

To incorporate linguistic information into the MDP formulation, we expand the action value function to include linguistic features. While state and action features are known at each point of computation, relevant words and their semantic roles are not observed. Therefore, we model text relevance as a hidden variable. Similarly, we use hidden variables to discriminate the words that describe *actions* and those that describe *state attributes* from the rest of the sentence. To incorporate these hidden variables in our action-value function, we model $Q(s, a)$ as a non-linear function approximation using a multi-layer neural network.

Despite the added complexity, all the parameters of our non-linear model can be effectively learnt in the Monte-Carlo Search framework. In Monte-Carlo Search, the action-value function is estimated by playing multiple simulated games starting at the current game state. We use the observed reward from these simulations to update the parameters of our neural network via backpropagation. This focuses learning on the current game state, allowing our method to learn language analysis and game-play appropriate to the observed game context.

Evaluation We test our method on the strategy game Civilization II, a notoriously challenging game with an immense action space.[2] As a source of knowledge for guiding our model, we use the official game manual. As a baseline, we employ a similar Monte-Carlo search based player which does not have access to textual information. We demonstrate that the linguistically-informed player significantly outperforms the baseline in terms of the number of games won. Moreover, we show that modeling the deeper linguistic structure of sentences further improves performance. In full-length games, our algorithm yields a 34% improvement over a language unaware baseline and wins over 65% of games against the built-in, hand-crafted AI of Civilization II.

[2]Civilization II was #3 in IGN's 2007 list of top video games of all time. (http://top100.ign.com/2007/ign_top_game_3.html)

Roadmap In Section 3.2, we provide intuition about the benefits of integrating textual information into learning algorithms for control. Section 3.3 describes prior work on language grounding, emphasizing the unique challenges and opportunities of our setup. This section also positions our work in a large body of research on Monte-Carlo based players. Section 3.4 presents background on Monte-Carlo Search as applied to game playing. In Section 3.5 we present a multi-layer neural network formulation for the action-value function that combines information from the text and the control application. Next, we present a Monte-Carlo method for estimating the parameters of this non-linear function. Sections 3.6 and 3.7 focus on the application of our algorithm to the game Civilization II. In Section 3.8 we compare our method against a range of competitive game-playing baselines, and empirically analyse the properties of the algorithm. Finally, in Section 3.9 we discuss the implications of this research, and conclude.

3.2 Learning Game Play from Text

In this section, we provide an intuitive explanation of how textual information can help improve action selection in a complex game. For clarity, we first discuss the benefits of textual information in the supervised scenario, thereby decoupling questions concerning modeling and representation from those related to parameter estimation. Assume that every state s is represented by a set of n features $[s_1, s_2, \ldots, s_n]$. Given a state s, our goal is to select the best possible action a_j from a fixed set A. We can model this task as multiclass classification, where each choice a_j is represented by a feature vector $[(s_1, a_j), (s_2, a_j), \ldots, (s_n, a_j)]$. Here, $(s_i, a_j), i \in [1, n]$ represents a feature created by taking the Cartesian product between $[s_1, s_2, \ldots, s_n]$ and a_j. To learn this classifier effectively, we need a training set that sufficiently covers the possible combinations of state features and actions. However, in domains with complex state spaces and a large number of possible actions, many instances of state-action feature values will be unobserved in training.

Now we show how the generalization power of the classifier can be improved using

textual information. Assume that each training example, in addition to a state-action pair, contains a sentence that may describe the action to be taken given the state attributes. Intuitively, we want to enrich our basic classifier with features that capture the correspondence between states and actions, and words that describe them. Given a sentence w composed of word types $w_1, w_2, \ldots, w_m$, these features can be of the form (s_i, w_k) and (a_j, w_k) for every $i \in [1, n]$, $k \in [1, m]$ and $a_j \in A$. Assuming that an action is described using similar words throughout the guide, we expect that a text-enriched classifier would be able to learn this correspondence via the features (a_j, w_k). A similar intuition holds for learning the correspondence between state-attributes and their descriptions represented by features (s_i, w_k). Through these features, the classifier can connect state s and action a_j based on the evidence provided in the guiding sentence and their occurrences in other contexts throughout the training data. A text-free classifier may not support such an association if the action does not appear in a similar state context in a training set.

The benefits of textual information extend to models that are trained using control feedback rather than supervised data. In this training scenario, the algorithm assesses the goodness of a given state-action combination by simulating a limited number of game turns after the action is taken and observing the control feedback provided by the underlying application. The algorithm has a built-in mechanism (see Section 3.4) that employs the observed feedback to learn feature weights, and intelligently samples the space in search for promising state-action pairs. When the algorithm has access to a collection of sentences, a similar feedback-based mechanism can be used to find sentences that match a given state-action pair (Section 3.5.1). Through the state- and action-description features (s_i, w_k) and (a_j, w_k), the algorithm jointly learns to identify relevant sentences and to map actions and states to their descriptions. Note that while we have used classification as the basis of discussion in this section, in reality our methods will learn a regression function.

3.3 Related Work

In this section, we first discuss prior work in the field of grounded language acquisition. Subsequently we look are two areas specific to our application domain – i.e., natural language analysis in the context of games, and Monte-Carlo Search applied to game playing.

3.3.1 Grounded Language Acquisition

Our work fits into the broad area of research on grounded language acquisition where the goal is to learn linguistic analysis from a non-linguistic situated context [52, 3, 66, 54, 79, 17, 80, 43, 10, 11, 74, 20, 71, 18, 44, 33]. The appeal of this formulation lies in reducing the need for manual annotations, as the non-linguistic signals can provide a powerful, albeit noisy, source of supervision for learning. In a traditional grounding setup it is assumed that the non-linguistic signals are parallel in content to the input text, motivating a machine translation view of the grounding task. An alternative approach models grounding in the control framework where the learner actively acquires feedback from the non-linguistic environment and uses it to drive language interpretation. Below we summarize both approaches, emphasizing the similarity and differences with our work.

Learning Grounding from Parallel Data In many applications, linguistic content is tightly linked to perceptual observations, providing a rich source of information for learning language grounding. Examples of such parallel data include images with captions [3], Robocup game events paired with a text commentary [17], and sequences of robot motor actions described in natural language [71]. The large diversity in the properties of such parallel data has resulted in the development of algorithms tailored for specific grounding contexts, instead of an application-independent grounding approach. Nevertheless, existing grounding approaches can be characterized along several dimensions that illuminate the connection between these algorithms:

- **Representation of Non-Linguistic Input** The first step in grounding words

in perceptual data is to discretize the non-linguistic signal (e.g., an image) into a representation that facilitates alignment. For instance, Barnard and Forsyth [3] segment images into regions that are subsequently mapped to words. Other approaches intertwine alignment and segmentation into a single step [54], as the two tasks are clearly interrelated. In our application, segmentation is not required as the state-action representation is by nature discrete.

Many approaches move beyond discretization, aiming to induce rich hierarchical structures over the non-linguistic input [27, 17, 18]. For instance, Fleischman and Roy [27] parse action sequences using a context-free grammar which is subsequently mapped into semantic frames. Chen and Mooney [17] represent action sequences using first order logic. In contrast, our algorithm capitalizes on the structure readily available in our data – state-action transitions. While inducing a richer structure on the state-action space may benefit mapping, it is a difficult problem in its own right from the field of hierarchical planning [5].

- **Representation of Linguistic Input** Early grounding approaches used the bag-of-words approach to represent input documents [79, 3, 27]. More recent methods have relied on a richer representation of linguistic data, such as syntactic trees [17] and semantic templates [71]. Our method incorporates linguistic information at multiple levels, using a feature-based representation that encodes both words as well as syntactic information extracted from dependency trees. As shown by our results, richer linguistic representations can significantly improve model performance.

- **Alignment** Another common feature of existing grounding models is that the training procedure crucially depends on how well words are aligned to non-linguistic structures. For this reason, some models assume that alignment is provided as part of the training data [27, 71]. In other grounding algorithms, the alignment is induced as part of the training procedure. Examples of such approaches are the methods of Barnard and Forsyth [3], and Liang et al. [43]. Both of these models jointly generate the text and attributes of the grounding

context, treating alignment as an unobserved variable.

In contrast, we do not explicitly model alignment in our model due to the lack of parallel data. Instead, we aim to extract relevant information from text and infuse it into a control application.

Learning Grounding from Control Feedback More recent work has moved away from the reliance on parallel corpora, using control feedback as the primary source of supervision. The assumption behind this setup is that when textual information is used to drive a control application, the application's performance will correlate with the quality of language analysis. It is also assumed that the performance measurement can be obtained automatically. This setup is conducive to reinforcement learning approaches which can estimate model parameters from the feedback signal, even it is noisy and delayed.

One line of prior work has focused on the task of mapping textual instructions into a *policy* for the control application, assuming that text fully specifies all the actions to be executed in the environment. For example, in our previous work [10, 11], this approach was applied to the task of translating instructions from a computer manual to executable GUI actions. Vogel and Jurafsky [74] demonstrate that this grounding framework can effectively map navigational directions to the corresponding path in a map. A second line of prior work has focused on full semantic parsing – converting a given text into a formal meaning representation such as first order logic [20]. These methods have been applied to domains where the correctness of the output can be accurately evaluated based on control feedback – for example, where the output is a database query which when executed provides a clean, oracle feedback signal for learning. This line of work also assumes that the text fully specifies the required output.

While our method is also driven by control feedback, our language interpretation task itself is fundamentally different. We assume that the given text document provides high-level advice without directly describing the correct actions for every potential game state. Furthermore, the textual advice does not necessarily translate

to a single strategy – in fact, the text may describe several strategies, each contingent on specific game states. For this reason, the strategy text cannot simply be interpreted directly into a policy. Therefore, our goal is to bias a learnt policy using information extracted from text. To this end, we do not aim to achieve a complete semantic interpretation, but rather use a partial text analysis to compute features relevant for the control application.

3.3.2 Language Analysis and Games

Even though games can provide a rich domain for situated text analysis, there have only been a few prior attempts at leveraging this opportunity [34, 26].

Eisenstein et al. [26] aim to automatically extract information from a collection of documents to help identify the rules of a game. This information, represented as predicate logic formulae, is estimated in an unsupervised fashion via a generative model. The extracted formulae, along with observed traces of game play are subsequently fed to an Inductive Logic Program, which attempts to reconstruct the rules of the game. While at the high-level, our goal is similar, i.e., to extract information from text useful for an external task, there are several key differences. Firstly, while Eisenstein et al. [26] analyze the text and the game as two disjoint steps, we model both tasks in an integrated fashion. This allows our model to learn a text analysis pertinent to game play, while at the same time using text to guide game play. Secondly, our method learns both text analysis and game play from a feedback signal inherent to the game, avoiding the need for pre-compiled game traces. This enables our method to operate effectively in complex games where collecting a sufficiently representative set of game traces can be impractical.

Gorniak and Roy [34] develop a machine controlled game character which responds to spoken natural language commands. Given traces of game actions manually annotated with transcribed speech, their method learns a structured representation of the text and aligned action sequences. This learnt model is then used to interpret spoken instructions by grounding them in the actions of a human player and the current game state. While the method itself does not learn to play the game, it enables human

control of an additional game character via speech. In contrast to Gorniak and Roy [34], we aim to develop algorithms to fully and autonomously control all actions of one player in the game. Furthermore, our method operates on the game's user manual rather than on human provided, contextually relevant instructions. This requires our model to identify if the text contains information useful in the current game state, in addition to mapping the text to productive actions. Finally, our method learns from game feedback collected via active interaction without relying on manual annotations. This allows us to effectively operate on complex games where collecting traditional labeled traces would be prohibitively expensive.

3.3.3 Monte-Carlo Search for Game AI

Monte-Carlo Search (MCS) is a state-of-the-art framework that has been very successfully applied, in prior work, to playing complex games such as Go, Poker, Scrabble, and real-time strategy games [29, 72, 7, 60, 57, 67, 2]. This framework operates by playing simulated games to estimate the goodness or *value* of different candidate actions. When the game's state and action spaces are complex, the number of simulations needed for effective play become prohibitively large. Previous application of MCS have addressed this issue using two orthogonal techniques: (1) they leverage domain knowledge to either guide or prune action selection, (2) they estimate the value of untried actions based on the observed outcomes of simulated games. This estimate is then used to bias action selection. Our MCS based algorithm for games relies on both of the above techniques. Below we describe the differences between our application of these techniques and prior work.

Leveraging Domain Knowledge Domain knowledge has been shown to be critically important to achieving good performance from MCS in complex games. In prior work this has been achieved by manually encoding relevant domain knowledge into the game playing algorithm – for example, via manually specified heuristics for action selection [7, 29], hand crafted features [72], and value functions encoding expert knowledge [67]. In contrast to such approaches, our goal is to automatically extract

and use domain knowledge from relevant natural language documents, thus bypassing the need for manual specification. Our method learns both text interpretation and game action selection based on the outcomes of simulated games in MCS. This allows it to identify and leverage textual domain knowledge relevant to the observed game context.

Estimating the Value of Untried Actions Previous approaches to estimating the value of untried actions have relied on two techniques. The first, *Upper Confidence bounds for Tree* (UCT) is a heuristic used in concert with the Monte-Carlo Tree Search variant of MCS. It augments an action's value with an exploration bonus for rarely visited state-action pairs, resulting in better action selection and better overall game performance [29, 67, 2]. The second technique is to learn a linear function approximation of action values for the current state s, based on game feedback [72, 63]. Even though our method follows the latter approach, we model action-value $Q(s, a)$ via a non-linear function approximation. Given the complexity of our application domain, this non-linear approximation generalizes better than a linear one, and as shown by our results significantly improves performance. More importantly, the non-linear model enables our method to represent text analysis as latent variables, allowing it to use textual information to estimate the value of untried actions.

3.4 Monte-Carlo Search

Our task is to leverage textual information to help us win a turn-based strategy game against a given opponent. In this section, we first describe the *Monte-Carlo Search* framework within which our method operates. The details of our linguistically informed Monte-Carlo Search algorithm are given in Section 3.5.

3.4.1 Game Representation

Formally, we represent the given turn-based stochastic game as a Markov Decision Process (MDP). This MDP is defined by the 4-tuple $\langle S, A, T, R \rangle$, where

- ***State space***, S, is the set of all possible states. Each state $s \in S$ represents a complete configuration of the game in-between player turns.

- ***Action space***, A, is the set of all possible actions. In a turn-based strategy game, a player controls multiple game units at each turn. Thus, each action $a \in A$ represents the joint assignment of all unit actions executed by the current player during the turn.

- ***Transition distribution***, $T(s' \mid s, a)$, is the probability that executing action a in state s will result in state s' at the next game turn. This distribution encodes the way the game state changes due to both the game rules, and the opposing player's actions. For this reason, $T(s' \mid s, a)$ is stochastic – as shown in Figure 3-2, executing the same action a at a given state s can result in different outcomes s'.

- ***Reward function***, $R(s) \in \mathbb{R}$, is the immediate reward received when transitioning to state s. The value of the reward correlates with the goodness of actions executed up to now, with higher reward indicating better actions.

All the above aspects of the MDP representation of the game – i.e., S, A, $T()$ and $R()$ – are defined implicitly by the game rules. At each step of the game, the game-playing agent can observe the current game state s, and has to select the best possible

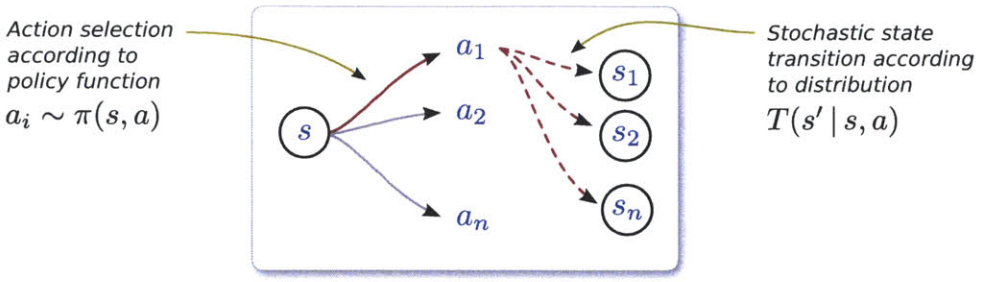


Figure 3-2: Markov Decision Process. Actions are selected according to *policy function* $\pi(s, a)$ given the current state s. The execution of the selected action a_i (e.g., a_1), causes the MDP to transition to a new state s' according to the stochastic state transition distribution $T(s' \mid s, a)$.

action a. When the agent executes action a, the game state changes according to the state transition distribution. While $T(s' \mid s, a)$ is not known a priori, state transitions can be sampled from this distribution by invoking the game code as a black-box simulator – i.e., by playing the game. After each action, the agent receives a reward according to the reward function $R(s)$. In a game playing setup, the value of this reward is an indication of the chances of winning the game from state s. Crucially, the reward signal may be delayed – i.e., $R(s)$ may have a non-zero value only for game ending states such as a win, a loss, or a tie.

The game playing agent selects actions according to a stochastic *policy* $\pi(s, a)$, which specifies the probability of selecting action a in state s. The expected total reward after executing action a in state s, and then following policy π is termed the *action-value function* $Q^{\pi}(s, a)$. Our goal is to find the *optimal policy* $\pi^*(s, a)$ which maximizes the expected total reward – i.e., maximizes the chances of winning the game. If the *optimal action-value function* $Q^{\pi^*}(s, a)$ is known, the optimal game-playing behaviour would be to select the action a with the highest $Q^{\pi^*}(s, a)$. While it may be computationally hard to find an optimal policy $\pi^*(s, a)$ or $Q^{\pi^*}(s, a)$, many well studied algorithms are available for estimating an effective approximation [68].

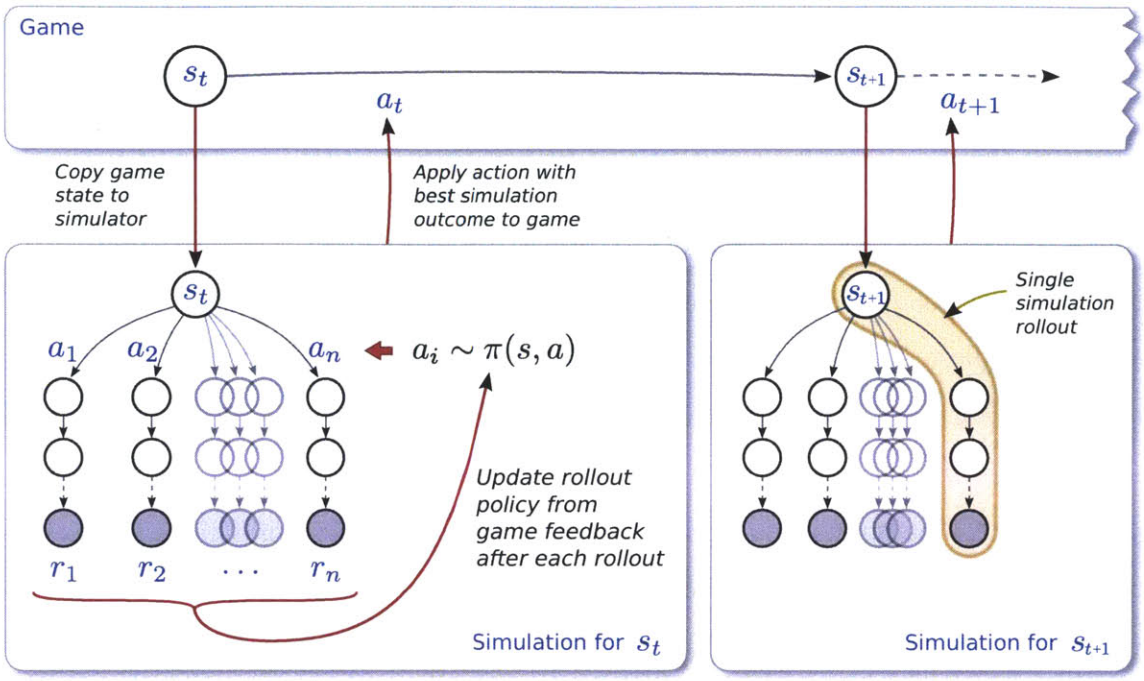


Figure 3-3: Overview of the Monte-Carlo Search algorithm. For each game state s_t, an independent set of simulated games or *roll-outs* are done to find the best possible game action a_t. Each roll-out starts at state s_t, with actions selected according to a *simulation policy* $\pi(s, a)$. This policy is learnt from the roll-outs themselves – with the roll-outs improving the policy, which in turn improves roll-out action selection. The process is repeated for every actual game state, with the simulation policy being relearnt from scratch each time.

3.4.2 Monte-Carlo Framework for Computer Games

The Monte-Carlo Search algorithm, shown in Figure 3-3, is a simulation-based search paradigm for dynamically estimating the action-values $Q^{\pi}(s, a)$ for a given state s_t (see Algorithm 1 for pseudo code). This estimate is based on the rewards observed during multiple *roll-outs*, each of which is a simulated game starting from state s_t.[3] Specifically, in each roll-out, the algorithm starts at state s_t, and repeatedly selects and

[3]Monte-Carlo Search assumes that it is possible to play simulated games. These simulations may be played against a heuristic AI player. In our experiments, the built-in AI of the game is used as the opponent.

executes actions according to a *simulation policy* $\pi(s, a)$, sampling state transitions from $T(s' \mid s, a)$. On game completion at time τ, the final reward $R(s_\tau)$ is measured, and the action-value function is updated accordingly.[4] As in Monte-Carlo control [68], the updated action-value function $Q^\pi(s, a)$ is used to define an improved simulation policy, thereby directing subsequent roll-outs towards higher scoring regions of the game state space. After a fixed number of roll-outs have been performed, the action with the highest average final reward in the simulations is selected and played in the actual game state s_t. This process is repeated for each state encountered during the actual game, with the action-value function being relearnt from scratch for each new game state.[5] The simulation policy usually selects actions to maximize the action-value function. However, sometimes other valid actions are also randomly explored in case they are more valuable than predicted by the current estimate of $Q^\pi(s, a)$. As the accuracy of $Q^\pi(s, a)$ improves, the quality of action selection improves and vice versa, in a cycle of continual improvement [68].

The success of Monte-Carlo Search depends on its ability to make a fast, local estimate of the action-value function from roll-outs collected via simulated play. However in games with large branching factors, it may not be feasible to collect sufficient roll-outs, especially when game simulation is computationally expensive. Thus it is crucial

[4]In general, roll-outs are run until game completion. If simulations are expensive, as is the case in our domain, roll-outs can be truncated after a fixed number of steps. This however depends on the availability of an approximate reward signal at the truncation point. In our experiments, we use the built-in score of the game as the reward. This reward is noisy, but available at every stage of the game.

[5]While it is conceivable that sharing the action-value function across the roll-outs of different game states would be beneficial, this was empirically not the case in our experiments. One possible reason is that in our domain, the game dynamics change radically at many points during the game – e.g., when a new technology becomes available. When such a change occurs, it may actually be detrimental to play according to the action-value function from the previous game step. Note however, that the action-value function is indeed shared across the roll-outs for a single game state s_t, with parameters updated by successive roll-outs. This is how the learnt model helps improve roll-out action selection, and thereby improves game play. The setup of relearning from scratch for each game state has been shown to be beneficial even in stationary environments [70].

that the learnt action-value function generalizes well from a small number of roll-outs – i.e., observed states, actions and rewards. One way to achieve this is to model the action-value function as a linear combination of state and action attributes:

$$Q^{\pi}(s,a) = \vec{w} \cdot \vec{f}(s,a).$$

Here $\vec{f}(s,a) \in \mathbb{R}^n$ is a real-valued feature function, and $\vec{w}$ is a weight vector. Prior work has shown such linear value function approximations to be effective in the Monte-Carlo Search framework [63].

Note that learning the action-value function $Q(s,a)$ in Monte-Carlo Search is related to Reinforcement Learning (RL) [68]. In fact, in our approach, we use standard gradient descent updates from RL to estimate the parameters of $Q(s,a)$. There is, however, one crucial difference between these two techniques: In general, the goal in RL is to find a $Q(s,a)$ applicable to *any* state the agent may observe during its existence. In the Monte-Carlo Search framework, the aim is to learn a $Q(s,a)$ specialized to the *current* state s. In essence, $Q(s,a)$ is relearnt for every observed state in the actual game, using the states, actions and feedback from simulations. While such relearning may seem suboptimal, it has two distinct advantages: first, since $Q(s,a)$ only needs to model the current state, it can be representationally much simpler than a global action-value function. Second, due to this simpler representation, it can be learnt from fewer observations than a global action-value function [70]. Both of these properties are important when the state space is extremely large, as is the case in our domain.

procedure PlayGame ()

Initialize game state to fixed starting state

$s_1 \leftarrow s_0$

for $t = 1 \ldots T$ **do**

  Run N simulated games

  for $i = 1 \ldots N$ **do**

    $(a_i, r_i) \leftarrow \text{SimulateGame}\,(s_t)$

  end

  Compute average observed utility for each action

  $a_t \leftarrow \arg\max_a \frac{1}{N_a} \sum_{i:a_i=a} r_i$

  Execute selected action in game

  $s_{t+1} \leftarrow T(s' \mid s_t, a_t)$

end

---

procedure SimulateGame (s_t)

for $u = t \ldots \tau$ **do**

  Compute Q function approximation

  $Q^\pi(s_u, a) = \vec{w} \cdot \vec{f}(s_u, a)$

  Sample action from action-value function in ϵ-greedy fashion:

  $$a_u \sim \pi(s_u, a) = \begin{cases} \text{uniform}\,(a \in A) & \text{with probability } \epsilon \\ \arg\max_a Q^\pi(s_u, a) & \text{otherwise} \end{cases}$$

  Execute selected action in game:

  $s_{u+1} \leftarrow T(s' \mid s_u, a_u)$

  if game is won or lost
    break

end

Update parameters $\vec{w}$ of $Q^\pi(s_t, a)$

Return action and observed utility:

return $a_t, R(s_\tau)$

Algorithm 1: The general Monte-Carlo algorithm.

3.5 Adding Linguistic Knowledge to the Monte-Carlo Framework

The goal of our work is to improve the performance of the Monte-Carlo Search framework described above, using information automatically extracted from text. In this section, we describe how we achieve this in terms of model structure and parameter estimation.

3.5.1 Model Structure

To achieve our aim of leveraging textual information to improve game-play, our method needs to perform three tasks: (1) identify sentences relevant to the current game state, (2) label sentences with a predicate structure, and (3) predict good game actions by combining game features with text features extracted via the language analysis steps. We first describe how each of these tasks can be modeled separately before showing how we integrate them into a single coherent model.

Modeling Sentence Relevance As discussed in Section 3.1, only a small fraction of a strategy document is likely to provide guidance relevant to the current game context. Therefore, to effectively use information from a given document d, we first need to identify the sentence y_i that is most relevant to the current game state s and action a.[6] We model this decision as a log-linear distribution, defining the probability of y_i being the relevant sentence as:

$$p(y = y_i|s, a, d) \propto e^{\vec{u}\cdot\vec{\phi}(y_i,s,a,d)}. \tag{3.1}$$

Here $\vec{\phi}(y_i, s, a, d) \in \mathbb{R}^n$ is a feature function, and $\vec{u}$ are the parameters we need to estimate. The function $\vec{\phi}(\cdot)$ encodes features that combine the attributes of sentence

[6] We use the approximation of selecting the single most relevant sentence as an alternative to combining the features of all sentences in the text, weighted by their relevance probability $p(y = y_i|s, a, d)$. This setup is computationally more expensive than the one used here.

y_i with the attributes of the game state and action. These features allow the model to learn correlations between game attributes and the attributes of relevant sentences.

Modeling Predicate Structure When using text to guide action selection, in addition to using word-level correspondences, we would also like to leverage information encoded in the structure of the sentence. For example, verbs in a sentence might be more likely to describe suggested game actions. We aim to access this information by inducing a task-centric predicate structure on the sentences. That is, we label the words of a sentence as either *action-description*, *state-description* or *background*. Given sentence y and its precomputed dependency parse q, we model the word-by-word labeling decision in a log-linear fashion – i.e., the distribution over the predicate labeling z of sentence y is given by:

$$\begin{aligned} p(z\,|y,q) &= p(\vec{e}\,|y,q) \\ &= \prod_j p(e_j|j,y,q), \\ p(e_j|j,y,q) &\propto e^{\vec{v}\cdot\vec{\psi}(e_j,j,y,q)}, \end{aligned} \tag{3.2}$$

where e_j is the predicate label of the j^{th} word. The feature function $\vec{\psi}(e_j, j, y, q) \in \mathbb{R}^n$, in addition to encoding word type and part-of-speech tag, also includes dependency parse information for each word. These features allow the predicate labeling decision to condition on the syntactic structure of the sentence.

Modeling the Action-Value Function Once the relevant sentence has been identified and labeled with a predicate structure, our algorithm needs to use this information along with the attributes of the current game state s to select the best possible game action a. To this end, we redefine the action-value function $Q(s, a)$ as a weighted linear combination of features of the game and the text information:

$$Q(s', a') = \vec{w} \cdot \vec{f}(s, a, y_i, z_i). \tag{3.3}$$

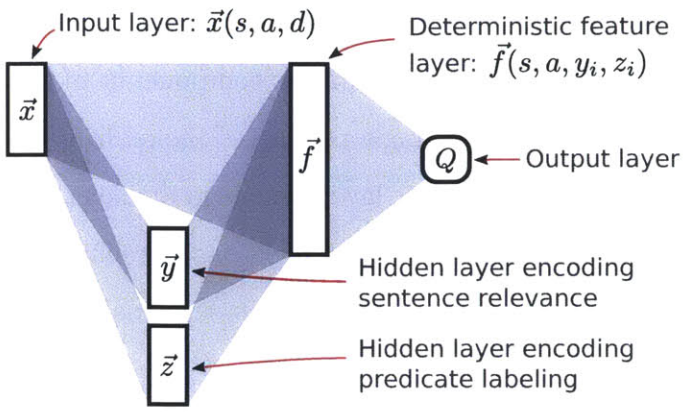


Figure 3-4: The structure of our neural network model. Each rectangle represents a collection of units in a layer, and the shaded trapezoids show the connections between layers. A fixed, real-valued feature function $\vec{x}(s, a, d)$ transforms the game state s, action a, and strategy document d into the input vector $\vec{x}$. The second layer contains two disjoint sets of hidden units $\vec{y}$ and $\vec{z}$, where $\vec{y}$ encodes the sentence relevance decisions, and $\vec{z}$ the predicate labeling. These are softmax layers, where only one unit is active at any time. The units of the third layer $\vec{f}(s, a, y_i, z_i)$ are a set of fixed real valued feature functions on s, a, d and the active units y_i and z_i of $\vec{y}$ and $\vec{z}$ respectively.

Here $s' = \langle s, d \rangle$, $a' = \langle a, y_i, z_i \rangle$, $\vec{w}$ is the weight vector, and $\vec{f}(s, a, y_i, z_i) \in \mathbb{R}^n$ is a feature function over the state s, action a, relevant sentence y_i, and its predicate labeling z_i. This structure of the action-value function allows it to explicitly learn the correlations between textual information, and game states and actions. The action a^* that maximizes $Q(s, a)$ is then selected as the best action for state s: [7]

$$a^* = \arg\max_a Q(s, a).$$

[7]Note that we select action a^* based on $Q(s, a)$, which depends on the relevant sentence y_i. This sentence itself is selected conditioned on action a. This may look like a cyclic dependency between actions and sentence relevance. However, that is not the case since $Q(s, a)$, and therefore sentence relevance $p(y|s, a, d)$, is computed for every candidate action $a \in A$. The actual game action a^* is then selected from this estimate of $Q(s, a)$.

Complete Joint Model The two text analysis models, and the action-value function described above form the three primary components of our text-aware game playing algorithm. We construct a single principled model from these components by representing each of them via different layers of the multi-layer neural network shown in Figure 3-4. Essentially, the text analysis decisions are modeled as latent variables by the second, hidden layer of the network, while the final output layer models the action-value function.

The *input layer* $\vec{x}$ of our neural network encodes the inputs to the model – i.e., the current state s, candidate action a, and document d. The *second layer* consists of two disjoint sets of hidden units $\vec{y}$ and $\vec{z}$, where each set operates as a stochastic 1-of-n softmax selection layer [14]. The activation function for units in this layer is the standard softmax function:

$$p(y_i = 1|\vec{x}) = e^{\vec{u}_i \cdot \vec{x}} \; / \; \sum_k e^{\vec{u}_k \cdot \vec{x}},$$

where y_i is the i^{th} hidden unit of $\vec{y}$, $\vec{u}_i$ is the weight vector corresponding to y_i, and k is the number of units in the layer. Given that this activation function is mathematically equivalent to a log-linear distribution, the layers $\vec{y}$ and $\vec{z}$ operate like log-linear models. Node activation in such a softmax layer simulates sampling from the log-linear distribution. We use layer $\vec{y}$ to replicate the log-linear model for sentence relevance from Equation (3.1), with each node y_i representing a single sentence. Similarly, each unit z_i in layer $\vec{z}$ represents a complete predicate labeling of a sentence, as in Equation (3.2).[8]

The third *feature layer* $\vec{f}$ of the neural network is deterministically computed given the active units y_i and z_i of the softmax layers, and the values of the input layer. Each unit in this layer corresponds to a fixed feature function $f_k(s, a, y_i, z_i) \in \mathbb{R}$. Finally the *output layer* encodes the action-value function $Q(s, a)$ as a weighted linear

[8]Our intention is to incorporate, into action-value function, information from only the most relevant sentence. Therefore, in practice, we only perform a predicate labeling of the sentence selected by the relevance component of the model.

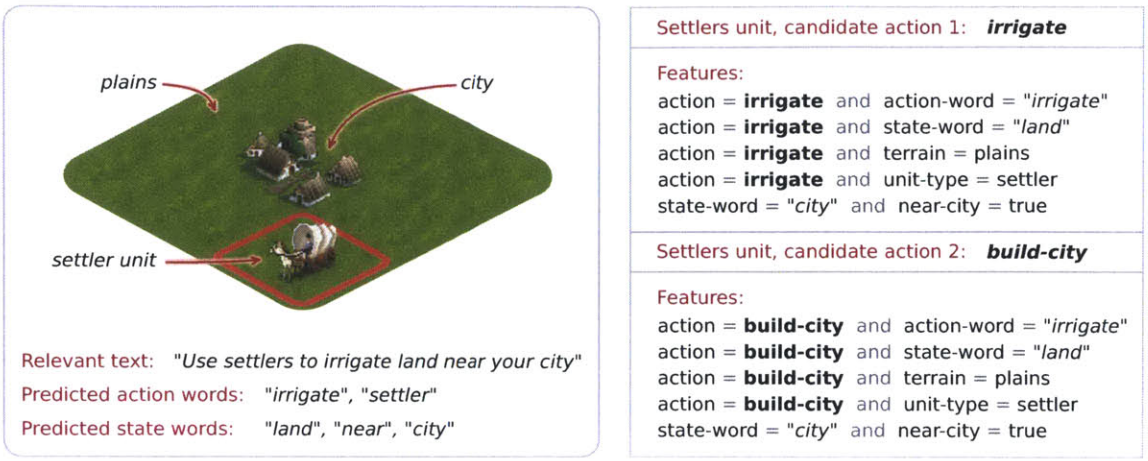


Figure 3-5: An example of text and game attributes, and the resulting candidate action features. On the left is a portion of a game state with arrows indicating game attributes. Also on the left is a sentence relevant to the game state along with action and state words identified by predicate labeling. On the right are two candidate actions for the *settler* unit along with the corresponding features. As mentioned in the relevant sentence, *irrigate* is the better of the two actions – executing it will lead to future higher game scores. This feedback and the features shown above allow our model to learn effective mappings – such as between the action-word "irrigate" and the action *irrigate*, and between state-word "city" and game attribute *near-city*.

combination of the units of the feature layer, thereby replicating Equation (3.3) and completing the joint model.

As an example of the kind of correlations learnt by our model, consider Figure 3-5. Here, a relevant sentence has already been selected for the given game state. The predicate labeling of this sentence has identified the words "irrigate" and "settler" as describing the action to take. When game roll-outs return higher rewards for the *irrigate* action of the *settler* unit, our model can learn an association between this action and the words that describe it. Similarly, it can learn the association between state description words and the feature values of the current game state – e.g., the word "city" and the binary feature *near-city*. This allows our method to leverage the automatically extracted textual information to improve game play.

3.5.2 Parameter Estimation

Learning in our method is performed in an online fashion: at each game state s_t, the algorithm performs a simulated game roll-out, observes the outcome of the simulation, and updates the parameters $\vec{u}$, $\vec{v}$ and $\vec{w}$ of the action-value function $Q(s_t, a_t)$. As shown in Figure 3-3, these three steps are repeated a fixed number of times at each actual game state. The information from these roll-outs is then used to select the actual game action. The algorithm relearns all the parameters of the action-value function for every new game state s_t. This specializes the action-value function to the subgame starting from s_t. Learning a specialized $Q(s_t, a_t)$ for each game state is common and useful in games with complex state spaces and dynamics, where learning a single global function approximation can be particularly difficult [70]. A consequence of this function specialization is the need for online learning – since we cannot predict which games states will be seen during testing, function specialization for those states cannot be done a priori, ruling out the traditional training/test separation.

Since our model is a non-linear approximation of the underlying action-value function of the game, we learn model parameters by applying non-linear regression to the observed final utilities from the simulated roll-outs. Specifically, we adjust the parameters by stochastic gradient descent, to minimize the mean-squared error between the action-value $Q(s, a)$ and the final utility $R(s_\tau)$ for each observed game state s and action a. The resulting update to model parameters θ is of the form:

$$\begin{aligned}\Delta\theta &= -\frac{\alpha}{2}\nabla_\theta \left[R(s_\tau) - Q(s,a)\right]^2 \\ &= \alpha \left[R(s_\tau) - Q(s,a)\right] \nabla_\theta Q(s,a;\theta),\end{aligned}$$

where α is a learning rate parameter. This minimization is performed via standard

error backpropagation [15, 56], resulting in the following online parameter updates:

$$\begin{aligned}
\vec{w} &\leftarrow \vec{w} + \alpha_w \left[Q - R(s_\tau)\right] \vec{f}(s, a, y_i, z_j), \\
\vec{u}_i &\leftarrow \vec{u}_i + \alpha_u \left[Q - R(s_\tau)\right] \hat{Q}\, \vec{x}\, [1 - p(y_i|\cdot)], \\
\vec{v}_i &\leftarrow \vec{v}_i + \alpha_v \left[Q - R(s_\tau)\right] \hat{Q}\, \vec{x}\, [1 - p(z_i|\cdot)].
\end{aligned}$$

Here α_w is the learning rate, $Q = Q(s, a)$, and $\vec{w}$, $\vec{u}_i$ and $\vec{v}_i$ are the parameters of the final layer, the sentence relevance layer and the predicate labeling layer respectively. The derivations of these update equations are given in Appendix B.1

3.5.3 Alternative Modeling Options

Reinforcement Learning One alternative to our Monte-Carlo algorithm is to learn the same model in a purely reinforcement learning framework. In this setup, the model parameters would be learnt by playing several independent training games, and once training is complete, the model would be applied to play test games. This approach requires that the learnt action-value function $Q(s, a)$ generalizes well across all the different states observed during a typical game. Given the diversity of game scenarios observed in Civilization II, the capacity of the model needs to be very high. In contrast, since Monte-Carlo search relearns model parameters for each game state, the model need only represent the current game state – thus significantly reducing the required model capacity. Essentially, Monte-Carlo search, as employed in our method, trades off higher simulation time for lower required model capacity.

Modeling Unit Coordination One of the weaknesses of our model is the explicit assumption that the actions of game units are independent of each other. While the simulations roll-outs implicitly model the interdependence of unit actions, the lack of explicit unit coordination significantly hampers our algorithm. However, fully modeling unit coordination is computationally intractable due to the branching factor of 10^{20}.

One potential solution is to leverage the inherently hierarchical relationships be-

tween units in Civilization II to model only the most useful aspects of coordination. For example, each player in Civilization II controls a single *nation*, which contains multiple *cities*. Each *city* is home to multiple units such as *workers* and *cavalry*. Thus we can condition action selection for *cities* on the action of the *nation*. Similarly, a *worker's* action selection could condition on the home *city's* actions. Such a hierarchical factorization of unit dependencies could allow the model to learn useful unit coordination while remaining computationally tractable. We leave such selective modeling of coordination as an avenue for future work.

3.6 Applying the Model

The game we test our model on, Civilization II, is a multi-player strategy game set either on Earth or on a randomly generated world. Each player acts as the ruler of one civilization, and starts with a few game units – i.e., two *Settlers*, two *Workers* and one *Explorer*. The goal is to expand your civilization by developing new technologies, building cities and new units, and to win the game by either controlling the entire world, or successfully sending a spaceship to another world. The map of the game world is divided into a grid of typically 4000 squares, where each grid location represents a tile of either land or sea. Figure 3-6 shows a portion of this world map from a particular instance of the game, along with the game units of one player. In our experiments, we consider a two-player game of Civilization II on a map of 1000 squares – the smallest map allowed on Freeciv. This map size is used by both novice human players looking for an easier game, as well as advanced players wanting a game of shorter duration. We test our algorithms against the built-in AI player of the game, with the difficulty level at the default *Normal* setting.[9]

3.6.1 Game States and Actions

We define the game state for Monte-Carlo search, to be the map of the game world, along with the attributes of each map tile, and the location and attributes of each player's cities and units. Some examples of these attributes are shown in Figure 3-7. The space of possible actions for each city and unit is defined by the game rules given the current game state. For example, cities can construct buildings such as harbors and banks, or create new units of various types; while individual units can move around on the grid, and perform unit specific actions such as *irrigation* for *Settlers*, and *military defense* for *Archers*. Since a player controls multiple cities and units,

[9]Freeciv has five difficulty settings: *Novice*, *Easy*, *Normal*, *Hard* and *Cheating*. As evidenced by discussions on the game's online forum (http://freeciv.wikia.com/index.php?title=Forum:Playing_Freeciv), some human players new to the game find even the *Novice* setting too hard.

Figure 3-6: A portion of the game map from one instance of a Civilization II game. Three cities, and several units of a single player are visible on the map. Also visible are the different terrain attributes of map tiles, such as grassland, hills, mountains and deserts.

Nation attributes:

- Amount of gold in treasury
- % of world controlled
- Number of cities
- Population
- Known technologies

City attributes:

- City population
- Surrounding terrain and resources
- Amount of food & resources produced
- Number of units supported by city
- Number & type of units present

Map tile attributes:

- Terrain type (e.g. grassland, mountain, etc)
- Tile resources (e.g. wheat, coal, wildlife, etc)
- Tile has river
- Construction on tile (city, road, rail, etc)
- Types of units (own or enemy) present

Unit attributes:

- Unit type (e.g., worker, explorer, archer, etc)
- Unit health & hit points
- Unit experience
- Is unit in a city?
- Is unit fortified?

Figure 3-7: Example attributes of game state.

the player's action space at each turn is defined by the combination of all possible actions for those cities and units. In our experiments, on average, a player controls approximately 18 units with each unit having 15 possible actions. The resulting action space for a player is very large – i.e., 10^{21}. To effectively deal with this large action space, we assume that given the state, the actions of each individual city and unit are independent of the actions of all other cities and units of the same player.[10] At the same time, we maximize parameter sharing by using a single action-value function for all the cities and units of the player.

3.6.2 Utility Function

Critically important to the Monte-Carlo search algorithm, is the availability of a utility function that can evaluate the outcomes of simulated game roll-outs. In the typical application of the algorithm, the final game outcome in terms of victory or loss is used as the utility function [72]. Unfortunately, the complexity of Civilization II, and the length of a typical game, precludes the possibility of running simulation roll-outs until game completion. The game, however, provides each player with a real valued *game score*, which is a noisy indicator of the strength of their civilization. Since we are playing a two-player game, our player's score relative to the opponent's can be used as the utility function. Specifically, we use the ratio of the game score of the two players.[11]

3.6.3 Features

All the components of our method operate on features computed over a basic set of text and game attributes. The text attributes include the words of each sentence along with their parts-of-speech and dependency parse information such as dependency types and parent words. The basic game attributes encode game information available

[10]Since each player executes game actions in turn, i.e. opposing units are fixed during an individual player's turn, the opponent's moves do not enlarge the player's action space.

[11]The difference between players' scores can also be used as the utility function. However, in practice the score ratio produced better empirical performance across all algorithms and baselines.

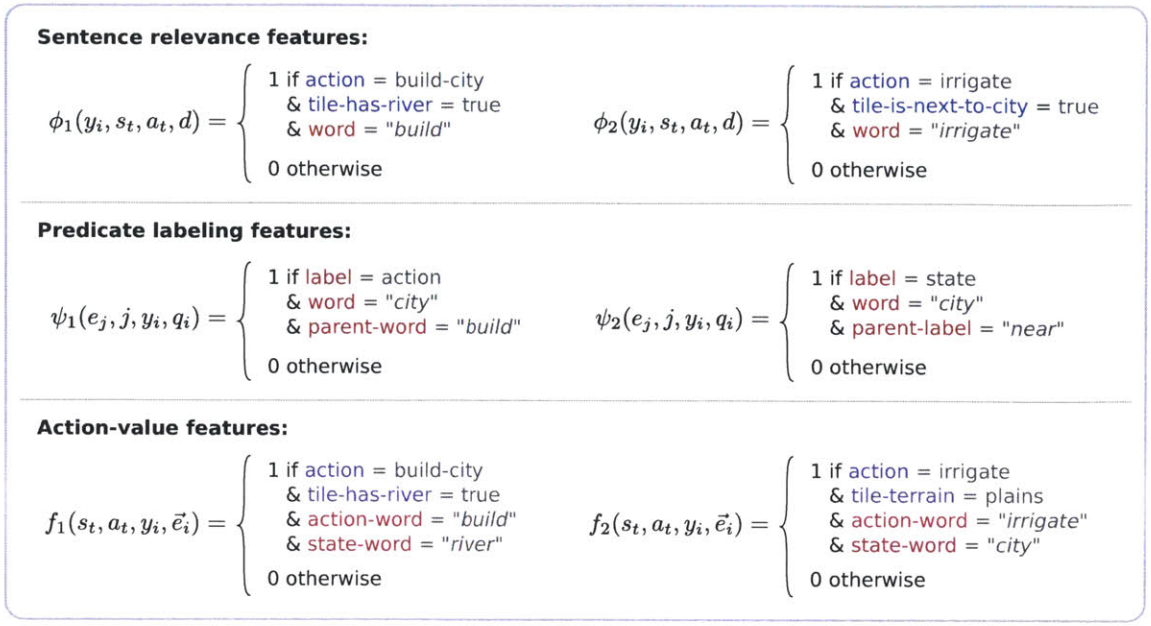


Figure 3-8: Some examples of the features used in our model. In each feature, conditions that test game attributes are highlighted in blue, and those that test words in the game manual are highlighted in red.

to human players via the game's graphical user interface. Some examples of these attributes are shown in Figure 3-7.

To identify the sentence most relevant to the current game state and candidate action, the sentence relevance component computes features over the combined basic attributes of the game and of each sentence from the text. These features $\vec{\phi}$, are of two types – the first computes the Cartesian product between the attributes of the game and the attributes of the candidate sentence. The second type consists of binary features that test for overlap between words from the candidate sentence, and the text labels of the current game state and candidate action. Given that only 3.2% of word tokens from the manual overlap with labels from the game, these similarity features are highly sparse. However, they serve as signposts to guide the learner – as shown by our results, our method is able to operate effectively even in the absence of these features, but performs better when they are present.

Predicate labeling, unlike sentence relevance, is purely a language task and as such operates only over the basic text attributes. The features for this component,

$\vec{\psi}$, compute the Cartesian product of the candidate predicate label with the word's type, part-of-speech tag, and dependency parse information. The final component of our model, the action-value approximation, operates over the attributes of the game state, the candidate action, the sentence selected as relevant, and the predicate labeling of that sentence. The features of this layer, $\vec{f}$, compute a three way Cartesian product between the attributes of the candidate action, the attributes of the game state, and the predicate labeled words of the relevant sentence. Overall, $\vec{\phi}$, $\vec{\psi}$ and $\vec{f}$ compute approximately 158,500, 7,900, and 306,800 features respectively – resulting in a total of 473,200 features for our full model. Figure 3-8 shows some examples of these features.

3.7 Experimental Setup

3.7.1 Datasets

We use the official game manual of Civilization II as our strategy guide document.[12] The text of this manual uses a vocabulary of 3638 word types, and is composed of 2083 sentences, each on average 16.9 words long. This manual contains information about the rules of the game, about the game user interface, and basic strategy advice about different aspects of the game. We use the Stanford parser [22], under default settings, to generate the dependency parse information for sentences in the game manual.

3.7.2 Experimental Framework

To apply our method to the Civilization II game, we use the game's open source reimplementation *Freeciv*.[13] We instrumented FreeCiv to allow our method to programmatically control the game – i.e., to measure the current game state, to execute game actions, to save/load the current game state, and to start and end games.[14]

Across all experiments, we start the game at the same initial state and run it for 100 steps. At each step, we perform 500 Monte-Carlo roll-outs. Each roll-out is run for 20 simulated game steps before halting the simulation and evaluating the outcome. Note that at each simulated game step, our algorithm needs to select an action for each game unit. Given an average number of units per player of 18, this results in 180,000 decisions during the 500 roll-outs. The pairing of each of these decisions with the corresponding roll-out outcome is used as a datapoint to update model parameters. We use a fixed learning rate of 0.0001 for all experiments. For our

[12] www.civfanatics.com/content/civ2/reference/Civ2manual.zip

[13] http://freeciv.wikia.com. Game version 2.2

[14] In addition to instrumentation, the code of FreeCiv (both the server and client) was changed to increase simulation speed by several orders of magnitude, and to remove bugs which caused the game to crash. To the best of our knowledge, the game rules and functionality are identical to the unmodified Freeciv version 2.2

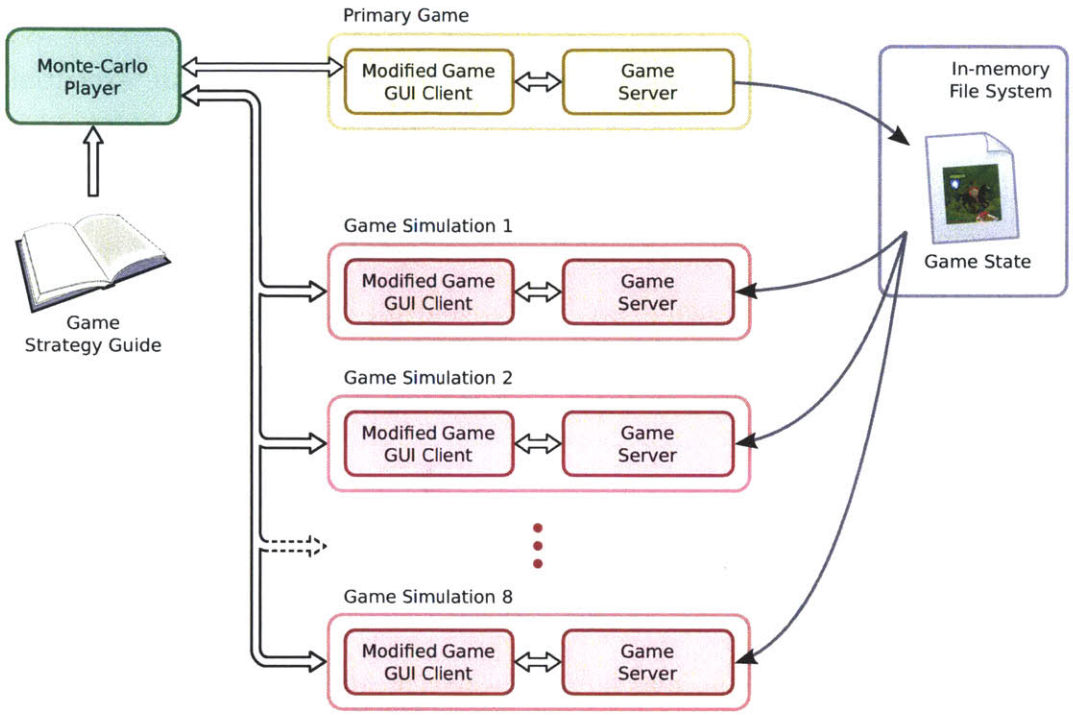


Figure 3-9: A diagram of the experimental framework, showing the Monte-Carlo player, the server for the primary game which the playing aims to win, and multiple game servers for simulated play. Communications between the multiple processes comprising the framework is via UNIX sockets and an in-memory file system.

method, and for each of the baselines, we run 200 independent games in the above manner, with evaluations averaged across the 200 runs. We use the same experimental settings across all methods, and all model parameters are initialized to zero.

Our experimental setup consists of our Monte-Carlo player, a primary game which we aim to play and win, and a set of simulation games. Both the primary game and the simulations are simply separate instances of the Freeciv game. Each instance of the Freeciv game is made up of one server process, which runs the actual game, and one client process, which is controlled by the Monte-Carlo player. At the start of each roll-out, the simulations are initialized with the current state of the primary game via the game save/reload functionality of Freeciv. Figure 3-9 shows a diagram of this experimental framework.

The experiments were run on typical desktop PCs with single Intel Core i7 CPUs (4 hyper-threaded cores per CPU). The algorithms were implemented to execute 8

simulation roll-outs in parallel by connecting to 8 independent simulation games. In this computational setup, approximately 5 simulation roll-outs are executed per second for our full model, and a single game of 100 steps runs in 3 hours. Since we treat the Freeciv game code as a black box, special care was taken to ensure consistency across experiments: all code was compiled on one specific machine, under a single fixed build environment (gcc 4.3.2); and all experiments were run under identical settings on a fixed set of machines running a fixed OS configuration (Linux kernel 2.6.35-25, libc 2.12.1).

3.7.3 Evaluation Metrics

We wish to evaluate two aspects of our method: how well it improves game play by leveraging textual information, and how accurately it analyzes text by learning from game feedback. We evaluate the first aspect by comparing our method against various baselines in terms of the percentage of games won against the built-in AI of Freeciv. This AI is a fixed heuristic algorithm designed using extensive knowledge of the game, with the intention of challenging human players.[15] As such, it provides a good open-reference baseline. We evaluate our method by measuring the percentage of games won, averaged over 100 independent runs. However, full games can sometimes last for multiple days, making it difficult to do an extensive analysis of model performance and contributing factors. For this reason, our primary evaluation measures the percentage of games won within the first 100 game steps, averaged over 200 independent runs. This evaluation is an underestimate of model performance – any game where the player has not won by gaining control of the entire game map within 100 steps is considered a loss. Since games can remain tied after 100 steps, two equally matched average players, playing against each other, will most likely have a win rate close to zero under this evaluation.

[15]While this AI is constrained to follow the rules of the game, it has access to information typically not available to human players, such as information about the technology, cities and units of it's opponents. Our methods on the other hand are restricted to the actions and information available to human players.

Method	% Win	% Loss	Std. Err.
Random	0	100	—
Built-in AI	0	0	—
Game only	17.3	5.3	± 2.7
Latent variable	26.1	3.7	± 3.1
Full model	**53.7**	5.9	± 3.5
Randomized text	40.3	4.3	± 3.4

Table 3.1: Win rate of our method and several baselines within the first 100 game steps, while playing against the built-in game AI. Games that are neither won nor lost are still ongoing. Our model's win rate is statistically significant against all baselines. All results are averaged across 200 independent game runs. The standard errors shown are for percentage wins.

3.8 Results

To adequately characterize the performance of our method, we evaluate it with respect to several different aspects. In this section, we first describe its game playing performance and analyze the impact of textual information. Then, we investigate the quality of the text analysis produced by our model in terms of both sentence relevance and predicate labeling.

3.8.1 Game Performance

Table 3.1 shows the performance of our method and several baselines on the primary 100-step evaluation. In this scenario, our language-aware Monte-Carlo algorithm wins on average 53.7% of games, substantially outperforming all baselines, while the best non-language-aware method has a win rate of only 26.1%. The dismal performance of the *Random* baseline and the game's own *Built-in AI*, playing against itself, are indications of the difficulty of winning games within the first 100 steps. As shown in Table 3.2, when evaluated on full length games, our method has a win rate of 65.4%

Method	% Wins	Standard Error
Game only	24.8	± 4.3
Latent variable	31.5	± 4.6
Full model	**65.4**	± 4.8

Table 3.2: Win rate of our method and two text-unaware baselines against the built-in AI. All results are averaged across 100 independent game runs.

compared to 31.5% for the best text-unaware baseline.[16]

Textual Advice and Game Performance To verify and characterize the impact of textual advice on our model's performance, we compare it against several baselines that do not have access to textual information. The simplest of these methods, *Game only*, models the action-value function $Q(s, a)$ as a linear approximation of the game's state and action attributes. This non-text-aware method wins only 17.3% of games (see Table 3.1). To confirm that our method's improved performance is not simply due to its inherently richer non-linear approximation, we also evaluate two ablative non-linear baselines. The first of these, *Latent variable* extends the linear action-value function of *Game only* with a set of latent variables. It is in essence a four layer neural network, similar to our full model, where the second layer's units are activated only based on game information. This baseline wins 26.1% of games (Table 3.1), significantly improving over the linear *Game only* baseline, but still trailing our text-aware method by more than 27%. The second ablative baseline, *Randomized text*, is identical to our model, except that it is given a randomly generated document as input. We generate this document by randomly permuting the locations of words

[16]Note that the performance of all methods on the full games is different from those published in Branavan et al. [12] and Branavan et al. [13]. These previously published numbers were biased by a code flaw in FreeCiv which caused the game to sporadically crash in the middle game play. While we originally believed the crash to be random, it was subsequently discovered to happen more often in losing games, and thereby biasing the win rates of all methods upwards. The numbers presented here are with this game bug fixed, with no crashes observed in any of the experiments.

in the game manual, thereby maintaining the document's statistical properties in terms of type frequencies. This ensures that the number of latent variables in this baseline is equal to that of our full model. Thus, this baseline has a model capacity equal to our text-aware method while not having access to any textual information. The performance of this baseline, which wins only 40.3% of games, confirms that information extracted from text is indeed instrumental to the performance of our method.

Figure 3-10 provides insight into how textual information helps improve game performance – it shows the observed game score during the Monte-Carlo roll-outs for our full model and the latent-variable baseline. As can be seen from this figure, the textual information guides our model to a high-score region of the search space far quicker than the non-text aware method, thus resulting in better overall performance. To evaluate how the performance of our method varies with the amount of available textual-information, we conduct an experiment where only random portions of the text are given to the algorithm. As shown in Figure 3-11, our method's performance varies linearly as a function of the amount of text, with the *Randomized text* experiment corresponding to the point where no information is available from text.

Impact of Seed Vocabulary on Performance The sentence relevance component of our model uses features that compute the similarity between words in a sentence, and the text labels of the game state and action. This assumes the availability of a seed vocabulary that names game attributes. In our domain, of the 256 unique text labels present in the game, 135 occur in the vocabulary of the game manual. This results in a sparse seed vocabulary of 135 words, covering only 3.7% of word types and 3.2% of word tokens in the manual. Despite this sparsity, the seed vocabulary can have a potentially large impact on model performance since it provides an initial set of word groundings. To evaluate the importance of this initial grounding, we test our method with an empty seed vocabulary. In this setup, our full model wins 49.0% of games, showing that while the seed words are important, our method can also operate effectively in their absence.

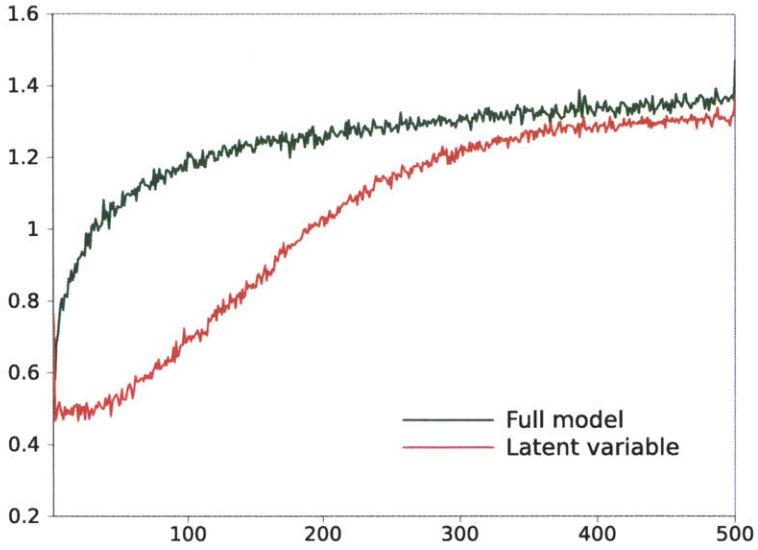


Figure 3-10: Observed game score as a function of Monte-Carlo roll-outs for our text-aware full model, and the text-unaware latent-variable model. Model parameters are updated after on each roll-out, and thus performance improves with roll-outs. As can be seen, our full model's performance improves dramatically over a small number of roll-outs, demonstrating the benefit it derives from textual information.

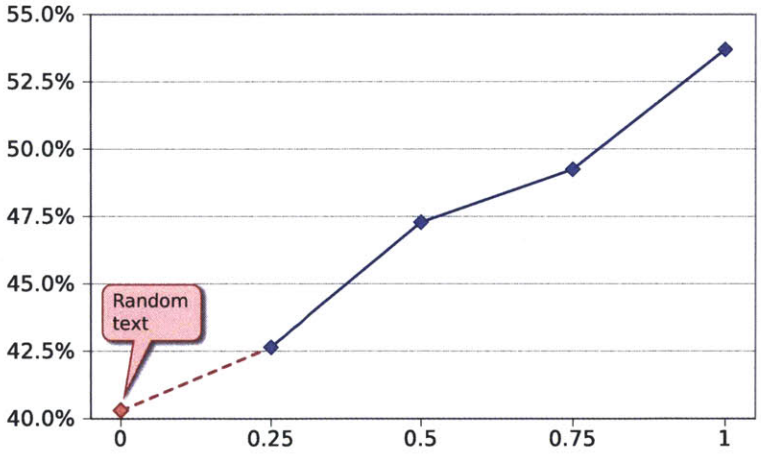


Figure 3-11: The performance of our text-aware model as a function of the amount of text available to it. We construct partial documents by randomly sub-sampling sentences from the full game manual. The x-axis shows the amount of sentences given to the method as a ratio of the full text. At the leftmost extreme is the performance of the *Randomized Text* baseline, showing how it fits into the performance trend at the point of having no useful textual information.

Method	% Win	% Loss	Std. Err.
Full model	**53.7**	5.9	± 3.5
Sentence relevance	46.7	2.8	± 3.5
No dependency information	39.6	3.0	± 3.4
No dependency label	50.1	3.0	± 3.5
No depend. parent POS tag	42.6	4.0	± 3.5
No depend. parent word	33.0	4.0	± 3.3

Table 3.3: Win rates of several ablated versions of our model, showing the contribution of different aspects of textual information to game performance. *Sentence relevance* is identical to the *Full model*, except that it lacks the predicate labeling component. The four methods at the bottom of the table ablate specific dependency features (as indicated by the method's name) from the predicate labeling component of the full model.

Linguistic Representation and Game Performance To characterize the contribution of language to game performance, we conduct a series of evaluations which vary the type and complexity of the linguistic analysis performed by our method. The results of this evaluation are shown in Table 3.3. The first of these, *Sentence relevance*, highlights the contributions of the two language components of our model. This algorithm, which is identical to our full model but lacks the predicate labeling component, wins 46.7% of games, showing that while it is essential to identify the textual advice relevant to the current game state, a deeper syntactic analysis of the extracted text substantially improves performance.

To evaluate the importance of dependency parse information in our language analysis, we vary the type of features available to the predicate labeling component of our model. The first of these ablative experiments, *No dependency information*, removes all dependency features – leaving predicate labeling to operate only on word type features. The performance of this baseline, a win rate of 39.6%, clearly shows that the dependency features are crucial for model performance. The remaining three meth-

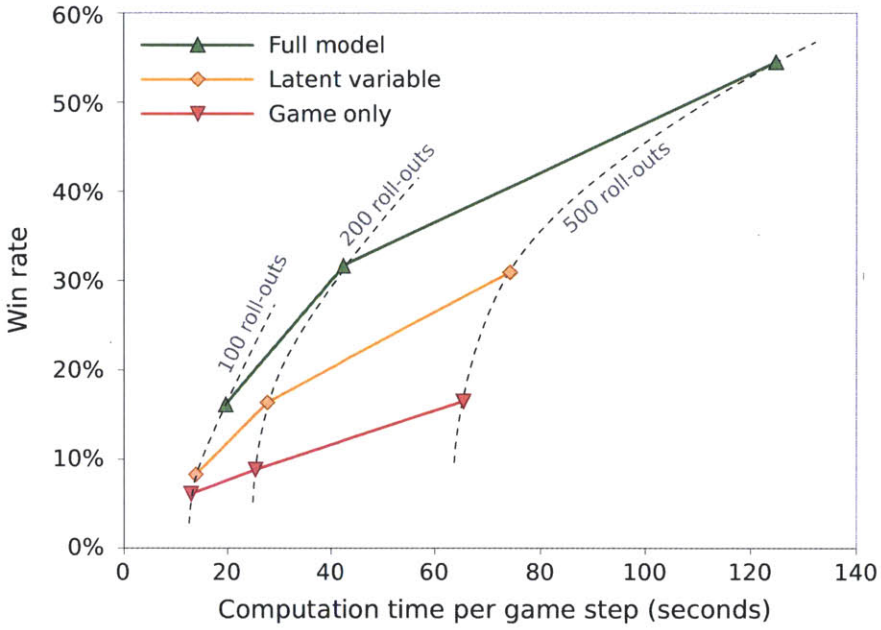


Figure 3-12: Win rate as a function of computation time per game step. For each Monte-Carlo search method, win rate and computation time were measured for 100, 200 and 500 roll-outs per game step, respectively.

ods – *No dependency label*, *No dependency parent POS tag* and *No dependency parent word* – each drop the dependency feature they are named after. The contribution of these features to model performance can be seen in Table 3.3.

Model Complexity vs Computation Time Trade-off One inherent disadvantage of non-linear models, when compared to simpler linear models, is the increase in computation time required for parameter estimation. In our Monte-Carlo Search setup, model parameters are re-estimated after each simulated roll-out. Therefore, given a fixed amount of time, more roll-outs can be done for a simpler and faster model. By its very nature, the performance of Monte-Carlo Search improves with the number of roll-outs. This trade-off between model complexity and roll-outs is important since a simpler model could compensate by using more roll-outs, and thereby outperform more complex ones. This scenario is particularly relevant in games where players have a limited amount of time for each turn.

To explore this trade-off, we vary the number of simulation roll-outs allowed for

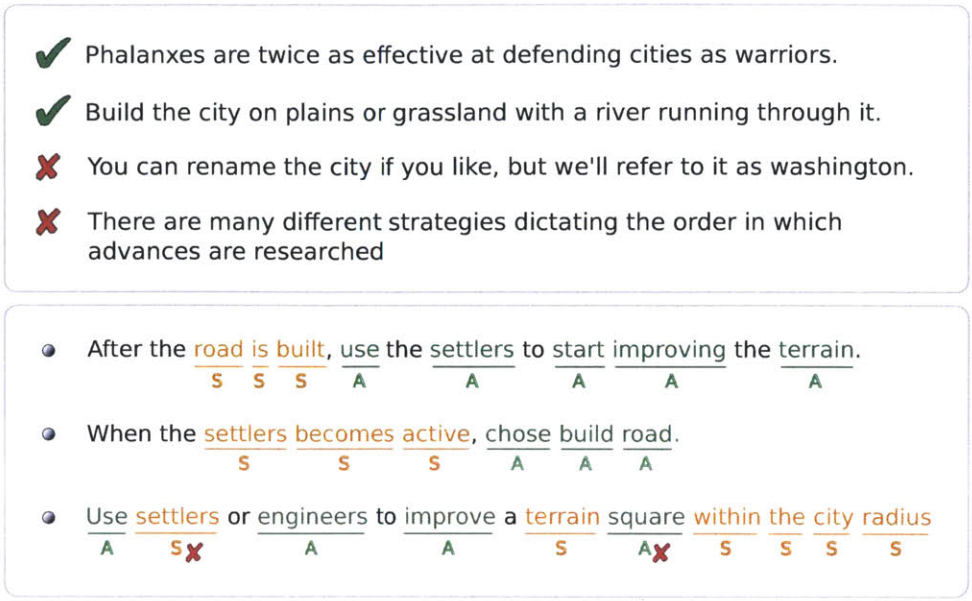


Figure 3-13: Examples of our method's sentence relevance and predicate labeling decisions. The box above shows two sentences (identified by green check marks) which were predicted as relevant, and two which were not. The box below shows the predicted predicate structure of three sentences, with "S" indicating *state* description,"A" *action* description and *background* words unmarked. Mistakes are identified with crosses.

each method at each game step, recording the win-rate and the average computation time per game. Figure 3-12 shows the results of this evaluation for 100, 200 and 500 roll-outs. While the more complex methods have higher computational demands, these results clearly show that even when given a fixed amount of computation time per game step, our text-aware model still produces the best performance by a wide margin.

Learned Game Strategy Qualitatively, all of the methods described here learn a basic *rush strategy*. Essentially, they attempt to develop basic technologies, build an army, and take over opposing cities as quickly as possible. The performance difference between the different models is essentially due to how well they learn this strategy.

There are two basic reasons why our algorithms learn the rush strategy. First, since we are attempting to maximize game score, the methods are implicitly biased

towards finding the fastest way to win – which happens to be the rush strategy when playing against the built-in AI of Civilization 2. Second, more complex strategies typically require the coordination of multiple game units. Since our models assume game units to be independent, they cannot explicitly learn such coordination – putting many complex strategies beyond the capabilities of our algorithms.

3.8.2 Accuracy of Linguistic Analysis

As described in Section 3.5, text analysis in our method is tightly coupled with game playing – both in terms of modeling, and in terms of learning from game feedback. We have seen from the results thus far, that this text analysis does indeed help game play. In this section we focus on the game-driven text analysis itself, and investigate how well it conforms to more common notions of linguistic correctness. We do this by comparing model predictions of sentence relevance and predicate labeling against manual annotations.

Sentence Relevance Figure 3-13 shows examples of the sentence relevance decisions produced by our method. To evaluate the accuracy of these decisions, we would ideally like to use a ground-truth relevance annotation of the game's user manual. This however, is impractical since the relevance decision is dependent on the game context, and is hence specific to each time step of each game instance. Therefore, we evaluate sentence relevance accuracy using a synthetic document. We create this document by combining the original game manual with an equal number of sentences which are known to be irrelevant to the game. These sentences are collected by randomly sampling from the Wall Street Journal corpus [47].[17] We evaluate sentence relevance on this synthetic document by measuring the accuracy with which game manual sentences are picked as relevant.

In this evaluation, our method achieves an average accuracy of 71.8%. Given that our model only has to differentiate between the game manual text and the Wall Street

[17]Note that sentences from the WSJ corpus contain words such as *city* which can potentially confuse our algorithm, causing it to select such sentences are relevant to game play.

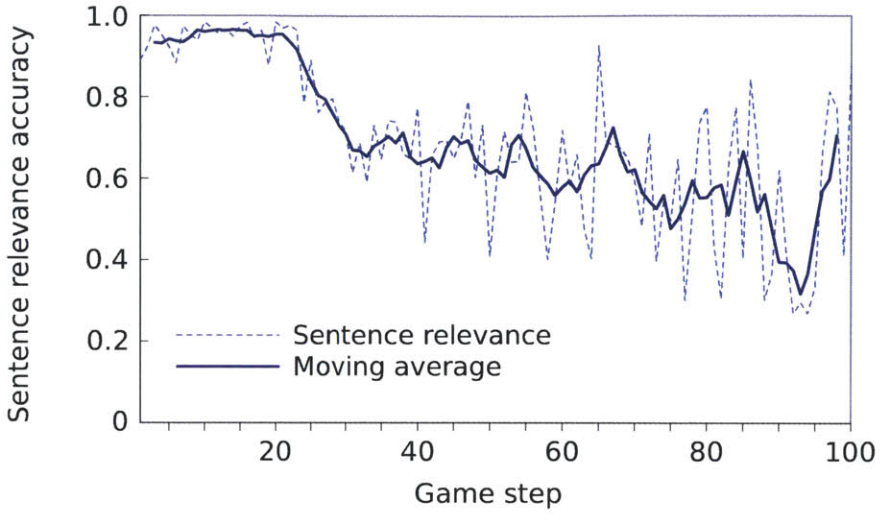


Figure 3-14: Accuracy of our method's sentence relevance predictions, averaged over 100 independent runs.

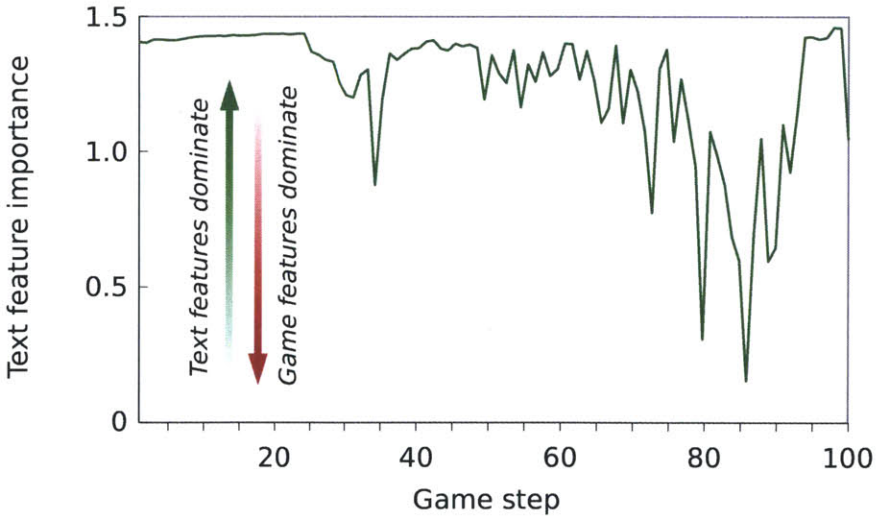


Figure 3-15: Difference between the norms of the text feature weights and game feature weights of the output layer of the neural network. Beyond the initial 25 steps of the game, our method relies increasingly on game features.

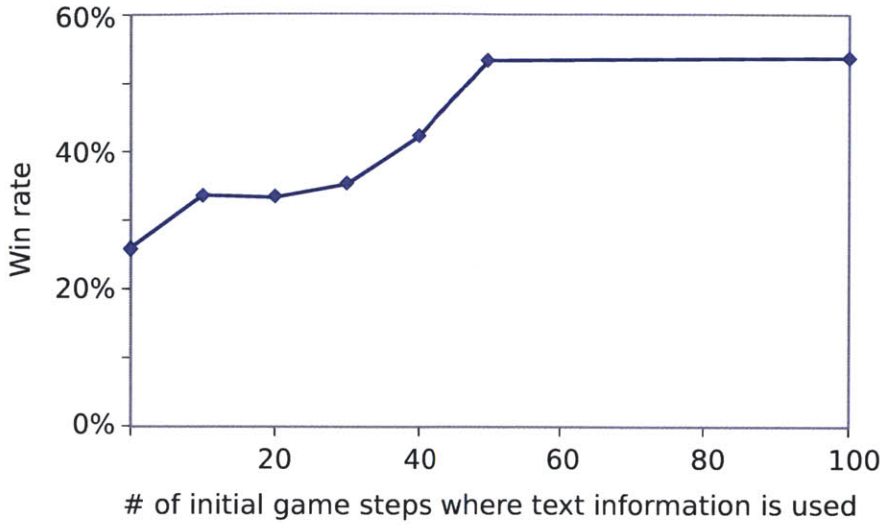


Figure 3-16: Graph showing how the availability of textual information during the initial steps of the game affects the performance of our full model. Textual information is given to the model for the first n steps (the x axis), beyond which point the algorithm has no access to text, and becomes equivalent to the *Latent Variable* model – i.e., the best non-text model.

Journal, this number may seem disappointing. Furthermore, as can be seen from Figure 3-14, the sentence relevance accuracy varies widely as the game progresses, with a high average of 94.2% during the initial 25 game steps. In reality, this pattern of high initial accuracy followed by a lower average is not entirely surprising: the official game manual for Civilization II is written for first time players. As such, it focuses on the initial portion of the game, providing little strategy advice relevant to subsequent game play.[18] If this is the reason for the observed sentence relevance trend, we would also expect the final layer of the neural network to emphasize game features over text features after the first 25 steps of the game. This is indeed the case, as can be seen in Figure 3-15.

To further test this hypothesis, we perform an experiment where the first n steps of the game are played using our full model, and the subsequent $100 - n$ steps are

[18]This is reminiscent of *opening books* for games like Chess or Go, which aim to guide the player to a playable middle game, without providing much information about subsequent game play.

Method	S/A/B	S/A
Random labeling	33.3%	50.0%
Model, first 100 steps	45.1%	78.9%
Model, first 25 steps	48.0%	92.7%

Table 3.4: Predicate labeling accuracy of our method and a random baseline. Column "S/A/B" shows performance on the three-way labeling of words as *state*, *action* or *background*, while column "S/A" shows accuracy on the task of differentiating between state and action words.

played without using any textual information. The results of this evaluation for several values of n are given in Figure 3-16, showing that the initial phase of the game is indeed where information from the game manual is most useful. In fact, this hybrid method performs just as well as our full model when $n = 50$, achieving a 53.3% win rate. This shows that our method is able to accurately identify relevant sentences when the information they contain is most pertinent to game play, and most likely to produce better game performance.

Predicate Labeling Figure 3-13 shows examples of the predicate structure output of our model. We evaluate the accuracy of this labeling by comparing it against a gold-standard annotation of the game manual.[19] Table 3.4 shows the performance of our method in terms of how accurately it labels words as *state*, *action* or *background*, and also how accurately it differentiates between *state* and *action* words. In addition to showing a performance improvement over the random baseline, these results display a clear trend: under both evaluations, labeling accuracy is higher during the initial stages of the game. This is to be expected since the model relies heavily on textual features during the beginning of the game (see Figure 3-15).

To verify the usefulness of our method's predicate labeling, we perform a final set

[19]Note that a ground truth labeling of words as either *action-description*, *state-description*, or *background* is based purely on the semantics of the sentence, and is independent of game state. For this reason, manual annotation is feasible, unlike in the case of sentence relevance.

game attribute	word
state: grassland	"city"
state: grassland	"build"
state: hills	"build"
action: settlers_build_city	"city"
action: set_research	"discovery"
action: settlers_build_city	"settler"
action: settlers_goto_location	"build"
action: city_build_barracks	"construct"
action: research_alphabet	"develop"
action: set_research	"discovery"

Figure 3-17: Examples of word to game attribute associations that are learnt via the feature weights of our model.

of experiments where predicate labels are selected uniformly at random within our full model. This random labeling results in a win rate of 44% – a performance similar to the *sentence relevance* model which uses no predicate information. This confirms that our method is able to identify a predicate structure which, while noisy, provides information relevant to game play. Figure 3-17 shows examples of how this textual information is grounded in the game, by way of the associations learnt between words and game attributes in the final layer of the full model. For example, our model learns a strong association between the game-state attribute *grassland* and the words "city" and "build", indicating that textual information about building cities maybe very useful when a player's unit is near grassland.

3.9 Conclusions

In this section we presented a novel approach for improving the performance of control applications by leveraging information automatically extracted from text documents, while at the same time learning language analysis based on control feedback. The model biases the learnt strategy by enriching the policy function with text features, thereby modeling the mapping between words in a manual and state-specific action selection. To effectively learn this grounding, the model identifies text relevant to the current game state, and induces a predicate structure on that text. These linguistic decisions are modeled jointly using a non-linear policy function trained in the Monte-Carlo Search framework.

Empirical results show that our model is able to significantly improve game win rate by leveraging textual information when compared to strong language-agnostic baselines. We also demonstrate that despite the increased complexity of our model, the knowledge it acquires enables it to sustain good performance even when the number of simulations is reduced. Moreover, deeper linguistic analysis, in the form of a predicate labeling of text, further improves game play. We show that information about the syntactic structure of text is crucial for such an analysis, and ignoring this information has a large impact on model performance. Finally, our experiments demonstrate that by tightly coupling control and linguistic features, the model is able to deliver robust performance in the presence of the noise inherent in automatic language analysis.

4

Learning High-Level Planning from Text

In this chapter, we consider the task of inducing high-level plans for completing given goals by leveraging textual information about the world. Our method learns to interpret textual descriptions of precondition relationship between objects in the world. Given a planning goal, the extracted textual information is then used to predict a sequence of waypoints for achieving the goal. Despite the mismatch between the abstractions of human language and the granularity of planning primitives, our method is able to effectively extract the precondition relations from text. While learning based on planning feedback, the relation extraction performance of our method rivals a manually supervised alternative. Our planning algorithm is able to leverage the extracted relation information to induce effective high-level plans, outperforming strong baselines that do not have access to the text.

4.1 Introduction

Understanding action preconditions and effects is a basic step in modeling the dynamics of the world. For example, having seeds is a precondition for growing wheat. Not surprisingly, preconditions have been extensively explored in various sub-fields of AI. However, existing work on action models has largely focused on tasks and techniques specific to individual sub-fields with little or no interconnection between

Seeds planted in farmland will grow to become wheat which can be harvested.

(a)

Low Level Actions for: seeds → wheat	
step 1:	move from (0,0) to (1,0)
step 2:	move from (1,0) to (2,0)
step 3:	pickup tool: hoe
step 4:	plow land with hoe at: (2,0)
step 5:	plant seeds at: (2,0)
...	
step N-1:	pickup tool: shears
step N:	harvest wheat with shears at: (2,0)

(b)

Figure 4-1: Text description of a precondition and effect (a), and the low-level actions connecting the two (b).

them. In NLP, precondition relations have been studied in terms of the linguistic mechanisms that realize them, while in classical planning, these relations are viewed as representations of world dynamics. In this chapter, we bring these two parallel views together, grounding the linguistic realization of these relations in the semantics of planning operations.

The challenge and opportunity of this fusion comes from the mismatch between the abstractions of human language and the granularity of planning primitives. Consider, for example, text describing a virtual world such as Minecraft[1] and a formal description of that world using planning primitives. Due to the mismatch in granularity, even the simple relation between seeds and wheat described in the sentence in Figure 4-1a results in dozens of low-level planning actions in the world, as can be seen in Figure 4-1b. While text provides a high-level description of world dynamics, it does not provide sufficient details for successful plan execution. On the other hand,

[1]http://www.minecraft.net/

planning with low-level actions does not suffer from this limitation, but is computationally intractable for even moderately complex tasks. As a consequence, in many practical domains, planners rely on manually-crafted high-level abstractions to make search tractable [30, 41].

The central idea of our work is to express the semantics of precondition relations extracted from text in terms of planning operations. For instance the precondition relation between seeds and wheat described in the sentence in Figure 4-1a indicates that plans which involve obtaining wheat will likely need to first obtain seeds. The novel challenge of this view is to model grounding at the level of relations, in contrast to prior work which focused on object-level grounding. We build on the intuition that the validity of precondition relations extracted from text can be informed by the execution of a low-level planner.[2] This feedback can enable us to learn these relations without annotations. Moreover, we can use the learnt relations to a guide a high level planner and ultimately improve planning performance.

We implement these ideas in the reinforcement learning framework wherein our model jointly learns to predict precondition relations from text and to perform high-level planning guided by those relations. For a given planning task and a set of candidate relations, our model repeatedly predicts a sequence of subgoals where each subgoal specifies an attribute of the world that must be made true. It then asks the low-level planner to find a plan between each consecutive pair of subgoals in the sequence. The observed feedback — whether the low-level planner succeeded or failed at each step — is utilized to update the policy for both text analysis and high-level planning.

We evaluate our algorithm in the Minecraft virtual world, using a large collection of user-generated on-line documents. Our results demonstrate the strength of our relation extraction technique. While using planning feedback as its only source of supervision, it achieves a performance on par with that of a supervised SVM base-

[2]If a planner can find a plan to successfully obtain wheat after obtaining seeds then seeds is likely a precondition for wheat. Conversely if a planner obtains wheat without first obtaining seeds then it is likely not a precondition.

line. Specifically, it yields an F-score of 66% compared to the 65% of the baseline. In addition we show that these extracted relations can be used to improve the performance of a high-level planner. As baselines for this evaluation, we employ the Metric-FF planner [36],[3] as well as a text-unaware variant of our model. Our results show that our text-driven high-level planner significantly outperforms all baselines in terms of completed planning tasks – it successfully solves 80% as compared to 41% for the Metric-FF planner and 69% for the text unaware variant of our model. In fact, the performance of our method approaches that of an oracle planner which uses manually-annotated preconditions.

[3]the state-of-art baseline used in the 2008 International Planning Competition. http://ipc.informatik.uni-freiburg.de/

4.2 Related Work

4.2.1 Extracting Event Semantics from Text

The task of extracting preconditions and effects has previously been addressed in the context of lexical semantics [62, 61]. These approaches combine large-scale distributional techniques with supervised learning to identify desired semantic relations in text. Such combined approaches have also been shown to be effective for identifying other relationships between events, such as causality [31, 8, 6, 16, 25].

Similar to these methods, our algorithm capitalizes on surface linguistic cues to learn preconditions from text. However, our only source of supervision is the feedback provided by the planning task which utilizes the predictions. Additionally, we not only identify these relations in text, but also show they are valuable in performing an external task.

4.2.2 Learning Semantics via Language Grounding

Our work fits into the broad area of grounded language acquisition, where the goal is to learn linguistic analysis from a situated context [52, 66, 79, 27, 50, 49, 10, 74, 43]. Within this line of work, we are most closely related to the reinforcement learning approaches that learn language by interacting with an external environment [10, 11, 74, 12].

The key distinction of our work is the use of grounding to learn abstract pragmatic relations, i.e. to learn linguistic patterns that describe relationships between objects in the world. This supplements previous work which grounds words to objects in the world [74, 12]. Another important difference of our setup is the way the textual information is utilized in the situated context. Instead of getting step-by-step instructions from the text, our model uses text that describes general knowledge about the domain structure. From this text it extracts relations between objects in the world which hold independent of any given task. Task-specific solutions are then constructed by a planner that relies on these relations to perform effective high-level

planning.

4.2.3 Hierarchical Planning

It is widely accepted that high-level plans that factorize a planning problem can greatly reduce the corresponding search space [51, 1]. Previous work in planning has studied the theoretical properties of valid abstractions and proposed a number of techniques for generating them [77, 38, 48, 4]. In general, these techniques use static analysis of the low-level domain to induce effective high-level abstractions. In contrast, our focus is on learning the abstraction from natural language. Thus our technique is complementary to past work, and can benefit from human knowledge about the domain structure.

4.3 Problem Formulation

Our task is two-fold. Given a text document which describes an environment, we wish to extract a set of precondition/effect relations implied by the text. We wish to then use these induced relations to determine an action sequence for completing a given task in this environment.

We formalize our task as illustrated in Figure 4-2. As input, we are given a world defined by the tuple $\langle S, A, T \rangle$, where S is the set of possible world states, A is the set of possible actions and T is the state transition function. Executing action a in state s causes a transition to a new state s' according to $T(s' \mid s, a)$. States are represented using first-order logic predicates $x_i \in X$, where each state is simply a set of such predicates, i.e. $s \subset X$.

The objective of the text analysis part of our task is to automatically extract a set of valid precondition/effect relationships from a given document d. Given our definition of the world state, preconditions and effects are merely single term predicates, x_i, in this world state. We assume that we are given a seed mapping between a predicate x_i, and the word types in the document that reference it (see Table 4.3 for examples). Thus, for each predicate pair $\langle x_k, x_l \rangle$, we want to utilize the text to predict whether x_k is a precondition for x_l; i.e., $x_k \rightarrow x_l$. For example, from the sentence in Figure 4-1, we want to predict that possessing seeds is a precondition for possessing wheat. Note that this relation is symmetric, and x_l can be interpreted as the effect of some sequence of actions performed on state x_k.

Each planning goal $g \in G$ is defined by a starting state s_0^g, and a final goal state s_f^g. This goal state is represented by a set of predicates which need to be made true. In the planning part of our task our objective is to find a sequence of actions $\vec{a}$ that connect s_0^g to s_f^g. Note that we assume that d does not contain step-by-step instructions for any individual task, but instead describes general facts about the given world that are useful for a wide variety of tasks.

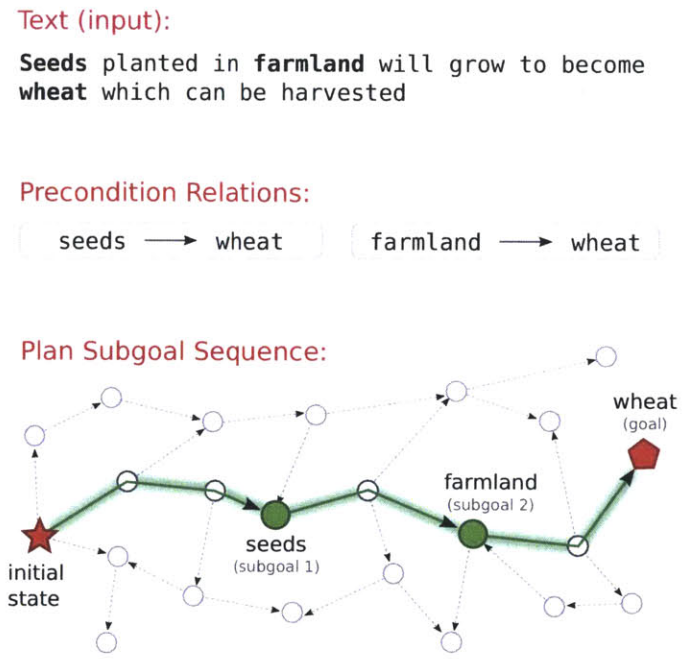


Figure 4-2: A high-level plan that shows two subgoals in a precondition relation. The corresponding sentence is shown above.

4.4 Model

The key idea behind our model is to leverage textual descriptions of preconditions and effects to guide the construction of high level plans. We define a high-level plan as a sequence of *subgoals*, where each subgoal is represented by a single-term predicate, x_i, that needs to be set in the corresponding world state – e.g. `have(wheat)=true`. Thus the set of possible subgoals is defined by the set of all possible single-term predicates in the domain. In contrast to low-level plans, the transition between these subgoals can involve multiple low-level actions. Our algorithm for textually informed high-level planning operates in four steps:

1. Use text to predict the preconditions of each subgoal. These predictions are for the entire domain and are not goal specific.

2. Given a planning goal and the induced preconditions, predict a subgoal sequence that achieves the given goal.

3. Execute the predicted sequence by giving each pair of consecutive subgoals to a low-level planner. This planner, treated as a black-box, computes the low-level plan actions necessary to transition from one subgoal to the next.

4. Update the model parameters, using the low-level planner's success or failure as the source of supervision.

We formally define these steps below.

4.4.1 Modeling Precondition Relations

Given a document d, and a set of subgoal pairs $\langle x_i, x_j \rangle$, we want to predict whether subgoal x_i is a precondition for x_j. We assume that precondition relations are generally described within single sentences. We first use our seed grounding in a preprocessing step where we extract all predicate pairs where both predicates are mentioned in the same sentence. We call this set the *Candidate Relations*. Note that this set

will contain many invalid relations since co-occurrence in a sentence does not imply a precondition relation — e.g. in Figure 4-2, seeds and farmland do not have a precondition relation despite co-occurring in the same sentence.[4] Thus for each sentence, $\vec{w}_k$, associated with a given *Candidate Relation*, $x_i \rightarrow x_j$, our task is to predict whether the sentence indicates the relation. We model this decision via a log linear distribution as follows:

$$p(x_i \rightarrow x_j \mid \vec{w}_k, q_k; \theta_c) \propto e^{\theta_c \cdot \phi_c(x_i, x_j, \vec{w}_k, q_k)}, \tag{4.1}$$

where θ_c is the vector of model parameters. We compute the feature function ϕ_c using our seed grounding, the sentence $\vec{w}_k$, and a given dependency parse q_k of the sentence. Given these per-sentence decisions, we predict the set of all valid precondition relations, C, in a deterministic fashion. We do this by considering a precondition $x_i \rightarrow x_j$ as valid if it is predicted to be valid by at least one sentence.

4.4.2 Modeling Subgoal Sequences

Given a planning goal g, defined by initial and goal states s_0^g and s_f^g, our task is to predict a sequence of subgoals $\vec{x}$ which will achieve the goal. We condition this decision on our predicted set of valid preconditions C, by modeling the distribution over sequences $\vec{x}$ as:

$$p(\vec{x} \mid s_0^g, s_f^g, C; \theta_x) = \prod_{t=1}^{n} p(x_t \mid x_{t-1}, s_0^g, s_f^g, C; \theta_x)$$

$$p(x_t \mid x_{t-1}, s_0^g, s_f^g, C; \theta_x) \propto e^{\theta_x \cdot \phi_x(x_t, x_{t-1}, s_0^g, s_f^g, C)}.$$

Here we assume that subgoal sequences are Markovian in nature and model individual subgoal predictions using a log-linear model.[5] Note that this model implicitly learns

[4] In our dataset only 11% of *Candidate.Relations* are valid.

[5] Note that these assumptions hold when the precondition relations among the subgoals form a linear chain. As discussed in Section 4.7, these strong assumptions are a reason why our model

precondition relations between subgoals x even in the absence of information from the text predictions C. This allows our method to effectively predict high-level plans even when the required precondition information is not available from text.

In contrast to Equation 4.1 where the predictions are goal-agnostic, the subgoal sequence predictions are goal-specific. As before, θ_x is the vector of model parameters, and ϕ_x is the feature function. Additionally, we assume a special stop symbol, $x_\emptyset$, which indicates the end of the subgoal sequence.

4.4.3 Parameter Update

Parameter updates in our model are done via reinforcement learning. Specifically, once the model has predicted a subgoal sequence for a given goal, the sequence is given to the low-level planner for execution. The success or failure of this execution is used to compute the reward signal r for parameter estimation. This predict-execute-update cycle is repeated until convergence. We assume that our reward signal r strongly correlates with the correctness of model predictions. Therefore, during learning, we need to find the model parameters that maximize expected future reward [68]. We perform this maximization via stochastic gradient ascent, using the standard policy gradient algorithm [69, 75].

We perform two separate policy gradient updates, one for each of the model components. The objective of the text component of our model is purely to predict the validity of preconditions. Therefore, subgoal pairs $\langle x_k, x_l \rangle$ where x_l is reachable from x_k, are given positive reward. The corresponding parameter update takes the following form:

$$\Delta\theta_c \leftarrow \alpha_c r \left[\phi_c(x_i, x_j, \vec{w}_k, q_k) - \mathbb{E}_{p(x_i \to x_j|\cdot)} \left[\phi_c(x_i, x_j, \vec{w}_k, q_k) \right] \right], \tag{4.2}$$

where α_c is the learning rate.

---

does not solve all the tasks in our experiments. These assumptions can potentially be removed by treating this portion of our task as a planning problem, instead of viewing it as a sequence prediction problem. We leave this as an avenue for future work.

The objective of the planning component of our model is to predict subgoal sequences that successfully achieve the given planning goals. Thus we directly use plan-success as a binary reward signal, which is applied to each subgoal decision in a sequence. This results in the following update:

$$\Delta\theta_x \leftarrow \alpha_x \, r \sum_t \left[\phi_x(x_t, x_{t-1}, s_0^g, s_f^g, C) - \mathbb{E}_{p(x_i'|\cdot)}\left[\phi_x(x_i', x_{t-1}, s_0^g, s_f^g, C)\right] \right], \quad (4.3)$$

where, t indexes into the subgoal sequence and α_x is the learning rate.

4.4.4 Alternative Modeling Options

Inducing High-level Actions Our method, described in the section above, views a high-level plan as a sequence of subgoals. An alternative approach to planning in complex domains is to induce high-level actions, and then plan using these actions. One way to construct such high-level actions is to identify repeating sequences of steps in observed low-level plans – since useful high-level actions are likely to occur often in actual plans. In this perspective, text that describes high-level tasks in terms of step-by-step instructions can be leveraged to connect the high-level action to it's low-level steps. A significant disadvantage of this approach, however, is that unlike our method it requires observations of useful low-level plans – since random exploration is highly unlikely to result in low-level action sequences that correspond to real high-level actions.

Planning Over Predicted Pairwise Relations One of the weaknesses of our method is the assumption that preconditions are Markovian, and that precondition relations among the subgoals for a given task form a linear chain. As discussed in Section 4.7, this limits the performance of our model. One way to overcome this deficiency is to apply traditional planning algorithms at the level of subgoals, using the precondition relations predicted by our model to define the state transition function. We leave this as an avenue for future work.

Input: A document d,

Set of planning tasks G,

Set of candidate precondition relations C_{all},

Reward function $r()$,

Number of iterations T

Initialization: Model parameters $\theta_x = 0$ and $\theta_c = 0$.

for $i = 1 \cdots T$ **do**

Sample valid preconditions:

$C \leftarrow \emptyset$

foreach $\langle x_i, x_j \rangle \in C_{all}$ **do**

foreach *Sentence* $\vec{w}_k$ *containing* x_i *and* x_j **do**

$v \sim p(x_i \rightarrow x_j \mid \vec{w}_k, q_k; \theta_c)$

if $v = 1$ **then** $C = C \cup \langle x_i, x_j \rangle$

end

end

Predict subgoal sequences for each task g.

foreach $g \in G$ **do**

Sample subgoal sequence $\vec{x}$ *as follows:*

for $t = 1 \cdots n$ **do**

Sample next subgoal:

$x_t \sim p(x \mid x_{t-1}, s_0^g, s_f^g, C; \theta_x)$

Construct low-level subtask from x_{t-1} to x_t

Execute low-level planner on subtask

end

Update subgoal prediction model using Eqn. 4.2

end

Update text precondition model using Eqn. 4.3

end

Algorithm 2: A policy gradient algorithm for parameter estimation in our model.

4.5 Applying the Model

We apply our method to Minecraft, a grid-based virtual world. Each grid location represents a tile of either land or water and may also contain resources. Users can freely move around the world, harvest resources and craft various tools and objects from these resources. The dynamics of the world require certain resources or tools as prerequisites for performing a given action, as can be seen in Figure 4-3. For example, a user must first craft a bucket before they can collect milk.

4.5.1 Defining the Domain

In order to execute a traditional planner on the Minecraft domain, we defined the domain using the Planning Domain Definition Language [28]. This is the standard task definition language used in the international planning competitions (IPC).[6] We define as predicates all aspects of the game state – for example, the location of resources in the world, the resources and objects possessed by the player, and the player's location. Our subgoals x_i and our task goals s_f^g map directly to these predicates. This results in a domain with significantly greater complexity than those solvable by traditional low-level planners. Table 4.1 compares the complexity of our domain with the planning domains used in the IPC.

4.5.2 Low-level Planner

As our low-level planner we employ the Metric-FF planner [36], the state-of-art baseline used in the 2008 International Planning Competition. Metric-FF is a forward-chaining heuristic state space planner. It's main heuristic is to simplify the task by ignoring operator delete lists. The number of actions in the solution for this simplified task is then used as the goal distance estimate for various search strategies.

[6] http://ipc.icaps-conference.org/

Domain	#Objects	#Pred Types	#Actions
Parking	49	5	4
Floortile	61	10	7
Barman	40	15	12
Minecraft	**108**	**16**	**68**

Table 4.1: A comparison of complexity between Minecraft and other domains used in the IPC-2011 sequential satisficing track. In Minecraft domain, the number of different objects, predicate types and actions is significantly larger.

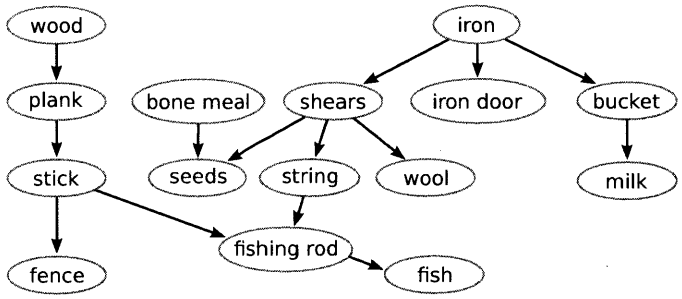


Figure 4-3: Examples of the precondition dependencies present in the Minecraft domain.

Words
Dependency Types
Dependency Type $\times$ Direction
Word $\times$ Dependency Type
Word $\times$ Dependency Type $\times$ Direction

Table 4.2: Example text features. A subgoal pair $\langle x_i, x_j \rangle$ is first mapped to word tokens using a small grounding table. Words and dependencies are extracted along paths between mapped target words. These are combined with path directions to generate the text features.

4.5.3 Features

The two components of our model leverage different types of information, and as a result, they each use a distinct set of features. The text component features ϕ_c are computed over sentences and their dependency parses. The Stanford parser [22] was used to generate the dependency parse information for each sentence. Examples of these features appear in Table 4.2. The sequence prediction component conditions on both the preconditions induced by the text component as well as the planning state and the previous subgoal. Thus ϕ_x contains features which check whether two subgoals are connected via an induced precondition relation, in addition to features which are simply the Cartesian product of domain predicates.

4.6 Experimental Setup

4.6.1 Datasets

As the text description of our virtual world, we use the Minecraft Wiki,[7] the most popular information source about the game. Our seed grounding of predicates contains 74 entries, examples of which can be seen in Table 4.3. We use this seed grounding to identify a set of 242 sentences that reference predicates in the Minecraft domain. This results in a set of 694 *Candidate Relations*. We also manually annotated the relations expressed in the text, identifying 94 of the *Candidate Relations* as valid. Additionally note that an average sentence in our corpus contains 20 words, and the whole corpus contains 979 unique word types.

We test our system on a set of 98 problems that involve collecting resources and constructing objects in the Minecraft domain – for example, fishing, cooking and making furniture. To assess the complexity of these tasks, we manually constructed high-level plans for these goals and solved them using the Metric-FF planner. On average, the execution of the sequence of low-level plans takes 35 actions, with 3 actions for the shortest plan and 123 actions for the longest. The average branching factor is 9.7, leading to an average search space of more than 10^{34} possible action sequences. For evaluation purposes we manually identify a set of *Gold Relations* consisting of all precondition relations that are valid in this domain, including those not discussed in the text.

4.6.2 Evaluation Metrics

We use our manual annotations to evaluate the accuracy of relation extraction. To evaluate our high-level planner, we use the standard measure adopted by the planning competitions. This evaluation measure simply assesses whether the planner completes a task in a predefined time.

[7] http://www.minecraftwiki.net/wiki/Minecraft_Wiki/

Domain Predicate	Noun Phrases
`have(plank)`	wooden plank wood plank
`have(stone)`	stone cobblestone
`have(iron)`	iron ingot

Table 4.3: Examples in our seed grounding table. Each predicate is mapped to one or more noun phrases that describe it in the text.

4.6.3 Baselines

To evaluate the performance of our relation extraction, we compare against an SVM classifier trained on gold standard precondition relations. We test the SVM baseline in a leave-one-out fashion.

To evaluate the performance of our text-aware high-level planner, we compare against five baselines. The first two baselines – *FF* and *No Text* – do not use any textual information. The *FF* baseline directly runs the Metric-FF planner on the given task, while the *No Text* baseline is a variant of our model that learns to plan in the reinforcement learning framework. It uses the same state-level features as our model, but does not have access to text.

The *All Text* baseline has access to the full set of 694 *Candidate Relations*. During learning, our full model refines this set of relations, while in contrast the *All Text* baseline always uses the full set.

The two remaining baselines constitute the upper bound on the performance of our model. The first, *Manual Text*, is a variant of our model which directly uses the links derived from manual annotations of preconditions in text. The second, *Gold*, has access to the *Gold Relations*. Note that the connections available to *Manual Text* are a subset of the *Gold* links, because the text does not specify all relations.

4.6.4 Experimental Details

All experimental results are averaged over 200 independent runs for both our model as well as the baselines. Each of these trials is run for 200 iterations with a maximum subgoal sequence length of 10. To find a low-level plan between each consecutive pair of subgoals, our high-level planner internally uses Metric-FF. We give Metric-FF a one-minute timeout to find such a low-level plan. To ensure that the comparison between the high-level planners and the FF baseline is fair, the FF baseline is allowed a runtime of 2,000 minutes. This is an upper bound on the time that our high-level planner can take over the 200 learning iterations, with subgoal sequences of length at most 10 and a one minute timeout. Lastly, during learning, we use a fixed learning rate of 0.0001 and we encourage our model to explore the state space by using the standard ϵ-greedy exploration strategy [68].

4.7 Results

4.7.1 Relation Extraction

Figure 4-4 shows the performance of our method on identifying preconditions in text. We also show the performance of the supervised SVM baseline. As can be seen, after 200 learning iterations, our model achieves an F-Measure of 66%, equal to the supervised baseline. These results support our hypothesis that planning feedback is a powerful source of information for text analysis. Figure 4-5 shows some examples of the sentences and the corresponding extracted relations.

4.7.2 Planning Performance

As shown in Table 4.4 our text-enriched planning model outperforms the text-free baselines by more than 10%. Moreover, the performance improvement of our model over the *All Text* baseline demonstrates that the accuracy of the extracted text relations does indeed impact planning performance. A similar conclusion can be reached by comparing the performance of our model and the *Manual Text* baseline.

The performance of the low-level planner, FF, indicates the fundamental difficulty of the planning tasks used in our experiments. Note that all the planning tasks are solvable – indeed, given manually specified high-level plans, FF is able to compute the low-level plans for all tasks. The reason for FF's poor performance is due to the large search space of the Minecraft domain. In the tasks that it is unable to solve, FF runs out of either memory or the pre-specified time limit.

The difference in performance of 2.35% between *Manual Text* and *Gold* shows the importance of the precondition information that is missing from the text. Note that *Gold* itself does not complete all tasks – this is largely because the Markov assumption made by our model does not hold for all tasks.[8]

[8]When a given task has two non-trivial preconditions, our model will choose to satisfy one of the two first, and the Markov assumption blinds it to the remaining precondition, preventing it from determining that it must still satisfy the other.

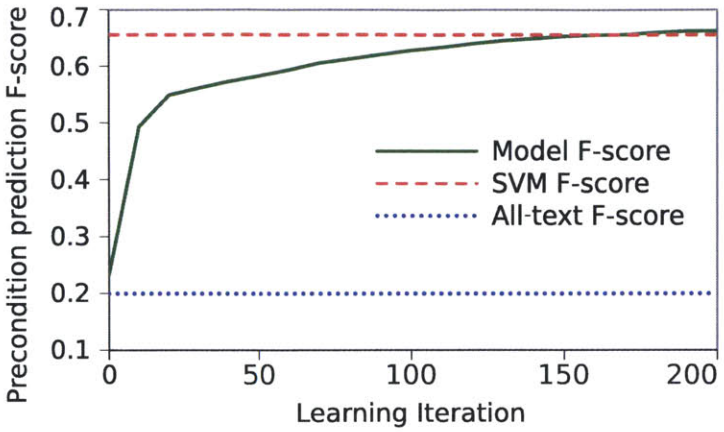


Figure 4-4: The performance of our model and a supervised SVM baseline on the precondition prediction task. Also shown is the F-Score of the full set of candidate relations which is used unmodified by *All Text*, and given as input to our model. The F-score of our model, averaged over 200 trials, is shown with respect to learning iterations.

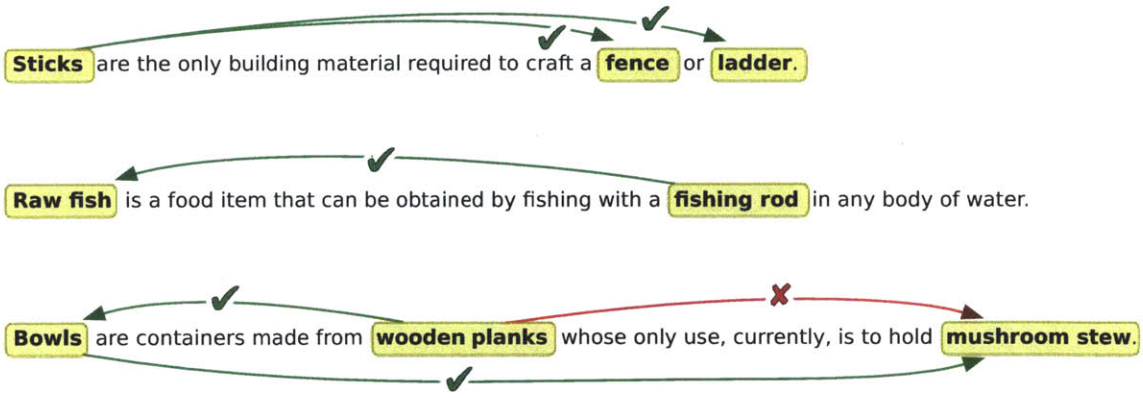


Figure 4-5: Some examples of the precondition relations predicted by our model from text. Check marks (✓) indicate correct predictions, while a cross (✘) marks the incorrect one. Note that each pair of highlighted noun phrases in a sentence is a candidate relation, and pairs that are not connected by an arrow were predicted to be invalid by our model.

Method	%Plans
FF	40.8
No text	69.4
All text	75.5
Full model	**80.2**
Manual text	84.7
Gold connection	87.1

Table 4.4: Percentage of tasks solved successfully by our model and the baselines. All performance differences between methods are statistically significant at $p \leq .01$.

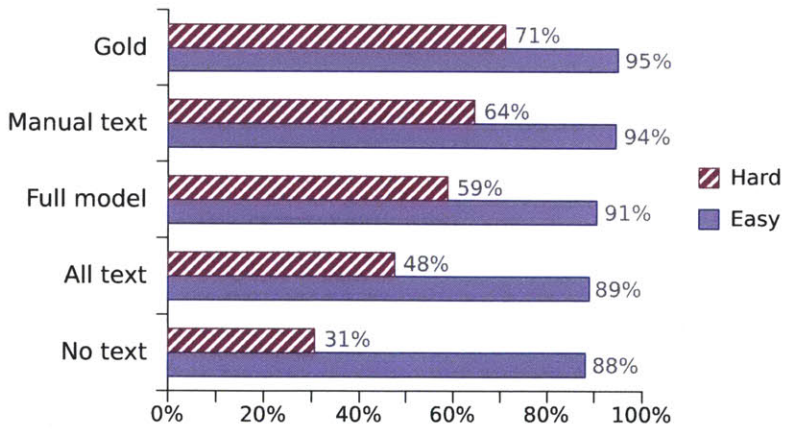


Figure 4-6: Percentage of problems solved by various models on Easy and Hard problem sets.

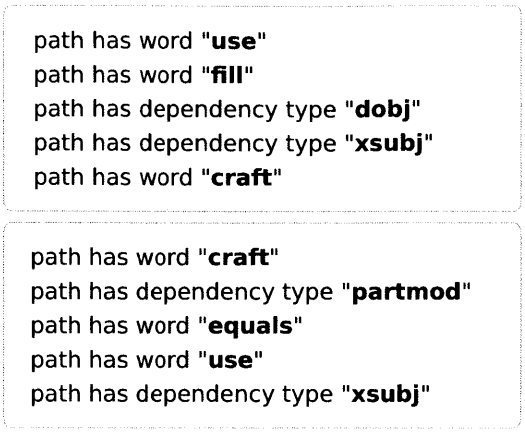


Figure 4-7: The top five positive features on words and dependency types learnt by our model (above) and by SVM (below) for precondition prediction.

Figure 4-6 breaks down the results based on the difficulty of the corresponding planning task. We measure problem complexity in terms of low-level steps needed to implement a manually constructed high-level plan. Based on this measure, we divide the problems into two sets. As can be seen, all of the high-level planners solve almost all of the easy problems. However, the performance varies greatly on the more challenging tasks, directly correlating with planner sophistication. On these tasks our model outperforms the *No Text* baseline by 28% and the *All Text* baseline by 11%.

4.7.3 Feature Analysis

Figure 4-7 shows the top ten positive features for our model and the SVM baseline. Both models picked up on the words that indicate precondition relations in this domain. For instance, the word *use* often occurs in sentences that describe the resources required to make an object, such as "bricks are items used to craft brick blocks". In addition to lexical features, dependency information is also given high weight by both learners. An example of this is a feature that checks for the direct object dependency type. This analysis is consistent with prior work on event semantics which shows lexico-syntactic features are effective cues for learning text relations [8, 6, 25].

4.8 Conclusions

In this chapter, we presented a novel technique for inducing precondition relations from text by grounding them in the semantics of planning operations. While using planning feedback as it's only source of supervision, our method for relation extraction achieves a performance on par with that of a supervised baseline. Furthermore, relation grounding provides a new view on classical planning problems which enables us to create high-level plans based on language abstractions. We show that building high-level plans in this manner significantly outperforms traditional techniques in terms of task completion.

5

Conclusions

In this thesis, we have shown how the connection between the semantics of language and the dynamics of the world can be leveraged to learn grounded language analysis. In particular, connecting language to control applications enables a powerful new source of supervision for language analysis. As our results across multiple tasks have shown, feedback signals inherent to the control application can be used to learn effective language analysis. Notably, the performance of our methods, which learn only from control feedback, rival that of equivalent manually supervised methods. Our approach of learning language from such feedback is a radical departure from prior work, which have primarily relied on learning in either a manually supervised or an unsupervised setting.

Another equally important advantage of grounding language in control applications is the ability to guide control actions using information from text. As shown by our results, automatically extracted textual information can significantly improve control performance. Remarkably, our methods are able to achieve effective language analysis, and improve control performance while starting with little or no prior knowledge about either language or the control application.

Across all three tasks discussed in this thesis, the appropriate modeling of linguistic phenomena present in the text has been crucial to good performance. Specifically, we have shown the importance of modeling the situational relevance of text, the abstract relations expressed in text, and the dynamics of the world corresponding to the text descriptions.

5.1 Future Work

Several avenues of investigation arise from the work described in this thesis, some of which we discuss below.

- **Mixed-mode Language** The models we have described in this thesis operate effectively over text containing different kinds of information – i.e., imperative step-by-step instructions, high-level strategy descriptions and general information about domain dynamics. Text documents, however, often contain a combination of such phenomena. Thus, methods that are able to simultaneously handle such variety will be essential for leveraging written human knowledge in real-world control applications.

- **Document Level Grounding** A particular limitation of our methods is the assumption that any information we wish to extract from text is expressed in a single sentence. This is clearly not always the case, and removing this assumption would expand our ability to extract useful information from text. This is particularly important in the case of documents containing complex information about the control application – information that practically cannot be expressed in single self-contained sentences.

- **Abstraction Hierarchies** In this thesis, we have shown the importance of modeling abstract relations expressed in text. However, our algorithm is limited to a single level of abstraction. An intriguing line of future work is to learn a hierarchy of such abstractions. In relation to the high-level planning method described in Chapter 4, learning an abstraction hierarchy can enable the induction of planning hierarchies from text. Moreover, such hierarchies are also important in light of text describing complex domains, where multiple levels of abstraction are essential for compactly describing the domain.

Appendix A

Instruction Interpretation

A.1 Derivations of Parameter Updates

This section details the derivation of parameter updates for our model for instruction interpretation (see Section 2.3). Our objective during learning is to find the policy parameters that maximize the expected future reward (i.e., value function V_θ) over all training documents. In this setting, the gradient ascent parameter updates for a log-linear policy function can be derived in a straightforward fashion [73].

We begin by computing the derivative of the total value function (i.e., total expected future reward) w.r.t θ,

$$\begin{aligned} V_\theta(s) &= \mathbb{E}_{p(h|\theta)}\left[r(h)\right] \\ &= \sum_{h \in H} r(h) p(h \mid \theta). \end{aligned}$$

Here the summation is over the histories H observed while interpreting the training documents D, and $p(h \mid \theta)$ is the probability of observing history h under policy parameters θ. In particular,

$$p(h \mid \theta) = \prod_{t=0}^{n-1} p(s_{t+1} \mid s_t, a_t)\, p(a_t \mid s_t, \theta)$$

$$= \left[\prod_{t=0}^{n-1} p(s_{t+1} \mid s_t, a_t) \right] \left[\prod_{t=0}^{n-1} p(a_t \mid s_t, \theta) \right]$$

$$= F(h)\, G(h, \theta).$$

Differentiating V_θ with respect to θ,

$$\frac{\partial}{\partial \theta_k} V_\theta = \sum_{h \in H} r(h)\, F(h)\, \frac{\partial}{\partial \theta_k} G(h, \theta),$$

Where

$$\frac{\partial}{\partial \theta_k} G(h, \theta) = \frac{\partial}{\partial \theta_k} \left[\prod_{t=0}^{n-1} p(a_t \mid s_t, \theta) \right]$$

$$= \sum_{t=0}^{n-1} \left[\frac{\partial}{\partial \theta_k} p(a_t \mid s_t, \theta) \prod_{t' \neq t} p(a_{t'} \mid s_{t'}, \theta) \right]$$

$$= \sum_{t=0}^{n-1} \left[\frac{\frac{\partial}{\partial \theta_k} p(a_t \mid s_t, \theta)}{p(a_t \mid s_t, \theta)} \prod_{t'=0}^{n-1} p(a_{t'} \mid s_{t'}, \theta) \right]$$

$$= \prod_{t'=0}^{n-1} p(a_{t'} \mid s_{t'}, \theta) \sum_{t=0}^{n-1} \frac{\partial}{\partial \theta_k} \log p(a_t \mid s_t, \theta)$$

$$= G(h, \theta) \sum_{t=0}^{n-1} \frac{\partial}{\partial \theta_k} \log p(a_t \mid s_t, \theta).$$

Which gives us Equation 2.8:

$$\frac{\partial}{\partial \theta_k} V_\theta = \sum_{h \in H} r(h)\, F(h)\, G(h, \theta) \sum_{t=0}^{n-1} \frac{\partial}{\partial \theta_k} \log p(a_t \mid s_t, \theta)$$

$$= \sum_{h \in H} r(h)\, p(h \mid \theta) \sum_{t=0}^{n-1} \frac{\partial}{\partial \theta_k} \log p(a_t \mid s_t, \theta)$$

$$= E_{p(h|\theta)}\left[r(h) \sum_{t=0}^{n-1} \frac{\partial}{\partial \theta} \log p(a_t \mid s_t; \theta) \right].$$

To expand the partial derivative in the inner sum, we note that we have defined the policy as a log-linear distribution. I.e.,

$$p(a_t \mid s_t; \theta) = \frac{e^{\theta \cdot \vec{\phi}(a_t, s_t)}}{\sum_a e^{\theta \cdot \vec{\phi}(a, s_t)}},$$

$$\log p(a_t \mid s_t; \theta) = \theta \cdot \vec{\phi}(a_t, s_t) - \log \sum_{a \in A} e^{\theta \cdot \vec{\phi}(a, s_t)}.$$

Therefore,

$$\begin{aligned}
\frac{\partial}{\partial \theta_k} \log p(a_t \mid s_t, \theta) &= \frac{\partial}{\partial \theta_k}\left[\theta \cdot \vec{\phi}(a_t, s_t) - \log \sum_{a \in A} e^{\theta \cdot \vec{\phi}(a, s_t)} \right] \\
&= \phi_k(a_t, s_t) - \left(\frac{\frac{\partial}{\partial \theta_k} \sum_{a \in A} e^{\theta \cdot \vec{\phi}(a, s_t)}}{\sum_{a' \in A} e^{\theta \cdot \vec{\phi}(a', s_t)}} \right) \\
&= \phi_k(a_t, s_t) - \left(\frac{\sum_{a \in A} \frac{\partial}{\partial \theta_k} e^{\theta \cdot \vec{\phi}(a, s_t)}}{\sum_{a' \in A} e^{\theta \cdot \vec{\phi}(a', s_t)}} \right) \\
&= \phi_k(a_t, s_t) - \left(\frac{\sum_{a \in A} \phi_k(a, s_t)\, e^{\theta \cdot \vec{\phi}(a, s_t)}}{\sum_{a' \in A} e^{\theta \cdot \vec{\phi}(a', s_t)}} \right) \\
&= \phi_k(a_t, s_t) - \sum_{a \in A} \phi_k(a, s_t) \left(\frac{e^{\theta \cdot \vec{\phi}(a, s_t)}}{\sum_{a' \in A} e^{\theta \cdot \vec{\phi}(a', s_t)}} \right)
\end{aligned}$$

$$= \phi_k(a_t, s_t) - \sum_{a \in A} \phi_k(a, s_t)\, p(a \mid s_t, \theta)$$

$$= \phi_k(a_t, s_t) - \mathbb{E}_{p(a|s_t,\theta)}\left[\phi_k(a, s_t)\right]$$

Expanding Equation 2.8, this gives us

$$\frac{\partial}{\partial \theta_k} V_\theta = \sum_{h \in H} r(h)\, p(h \mid \theta) \sum_{t=0}^{n-1} \frac{\partial}{\partial \theta_k} \log p(a_t \mid s_t, \theta)$$

$$= \sum_{h \in H} r(h)\, p(h \mid \theta) \sum_{t=0}^{n-1} \left(\phi_k(a_t, s_t) - \sum_{a \in A} \phi_k(a, s_t)\, p(a \mid s_t, \theta) \right).$$

During training, we select actions a by sampling from the policy $p(a \mid s, \theta)$. The resulting observed histories h are therefore samples from $p(h \mid \theta)$. Under this action selection scheme, using a learning rate of α, we get the following stochastic gradient ascent parameter updates:

$$\Delta\theta = \sum_{h \in H} \alpha\, r(h) \sum_{t=0}^{n-1} \left(\phi_k(a_t, s_t) - \sum_{a \in A} \phi_k(a, s_t)\, p(a \mid s_t, \theta) \right).$$

When single history samples h are used to approximate the gradient, the parameter updates simplify to

$$\Delta\theta = \alpha\, r(h) \sum_{t=0}^{n-1} \left(\phi_k(a_t, s_t) - \sum_{a \in A} \phi_k(a, s_t)\, p(a \mid s_t, \theta) \right).$$

A.2 Features

Basic features

The following templates are used to compute basic features that are used by the model for interpreting low-level instructions as well as the model for interpreting high-level instructions:

- *Word W_c describes next environment command.*
- *Binned closest linear distance from word W_c to the words of the previous command.*
- *Word W_o describes GUI widget of next command.*
- *Word W_o describes GUI widget of next command, and W matches the label of some widget in the environment.*
- *Lowest edit distance of W_o to widget labels in the environment.*
- *Linear distance between candidate command word W_c and candidate GUI widget word W_o.*
- *Word W_p is the text parameter of candidate command.*
- *The word immediately before W_p is W.*
- *The word immediately after W_p is W.*
- *Linear distance between candidate parameter word W_p and candidate command word W_c.*
- *Linear distance between candidate parameter word W_p and candidate GUI widget word W_o.*
- *Word W is present in the shortest word span containing W_c, W_o and W_p.*
- *Number of sentences between words describing next command, and the words selected for the previous command.*
- *Edit distance of candidate GUI widget's label to W_o.*

- *Candidate GUI widget, O, is visible.*
- *Candidate GUI widget, O, has input focus.*
- *Candidate GUI widget, O, is in the foreground.*
- *Candidate GUI widget, O, has input enabled.*
- *Candidate GUI widget, O, is selected.*
- *Candidate GUI widget, O, is a child of another GUI widget.*
- *Candidate GUI widget, O, appeared on screen after previous environment command was executed.*
- *Candidate GUI widget, O, changed it's name after previous environment command was executed.*
- *Class name of candidate GUI widget is present in sentence immediately before or after W_o.*
- *Candidate command word is W_c and candidate command is C.*
- *Candidate command is C and candidate GUI widget's class is T.*

Look-ahead features

The following templates are used to compute the look-ahead features used by the model for interpreting high-level instructions. These features look at how the remaining text of the sentence can potentially be interpreted within the partial environment model, after the current candidate command is executed.

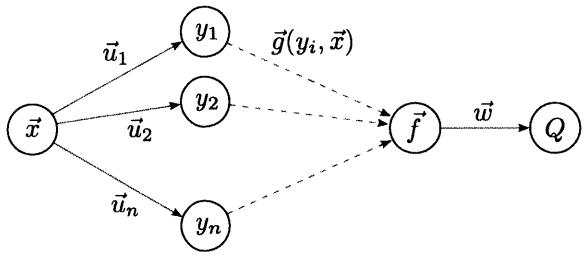
- *Ratio of unused words in sentence that can potentially be interpreted.*
- *Number of low-level environment commands involved in potential interpretation.*
- *Average reward received in potential interpretation.*

Appendix B

Strategy Interpretation

B.1 Derivations of Parameter Updates

The parameter of our model are estimated via standard error backpropagation [15, 56]. To derive the parameter updates, consider the slightly simplified neural network shown below. This network is identical to our model, but for the sake of clarity, it has only a single second layer $\vec{y}$ instead of the two parallel second layers $\vec{y}$ and $\vec{z}$. The parameter updates for these parallel layers $\vec{y}$ and $\vec{z}$ are similar, therefore we will show the derivation only for $\vec{y}$ in addition to the updates for the final layer.



As in our model, the nodes y_i in the network above are activated via a softmax function; the third layer, $\vec{f}$, is computed deterministically from the active nodes of the second layer via the function $\vec{g}(y_i, \vec{x})$; and the output Q is a linear combination

of $\vec{f}$ weighted by $\vec{w}$:

$$\begin{aligned} p(y_i = 1 \mid \vec{x}; \vec{u}_i) &= \frac{e^{\vec{u}_i \cdot \vec{x}}}{\sum_k e^{\vec{u}_k \cdot \vec{x}}}, \\ \vec{f} &= \sum_i \vec{g}(\vec{x}, y_i)\, p(y_i \mid \vec{x}; \vec{u}_i), \\ Q &= \vec{w} \cdot \vec{f}. \end{aligned}$$

Our goal is to minimize the mean-squared error e by gradient descent. We achieve this by updating model parameters along the gradient of e with respect to each parameter. Using θ_i as a general term to indicate our model's parameters, this update takes the form:

$$\begin{aligned} e &= \frac{1}{2}(Q - R)^2, \\ \Delta\theta_i &= \frac{\partial e}{\partial \theta_i} \\ &= (Q - R)\ \frac{\partial Q}{\partial \theta_i}. \end{aligned} \tag{B.1}$$

From Equation (B.1), the updates for final layer parameters are given by:

$$\begin{aligned} \Delta w_i &= (Q - R)\ \frac{\partial Q}{\partial w_i} \\ &= (Q - R)\ \frac{\partial}{\partial w_i}\, \vec{w} \cdot \vec{f} \\ &= (Q - R)\ f_i. \end{aligned}$$

Since our model samples the one most relevant sentence y_i, and the best predicate labeling z_i, the resulting online updates for the output layer parameters $\vec{w}$ are:

$$\vec{w} \leftarrow \vec{w} + \alpha_w\, [Q - R(s_\tau)]\, \vec{f}(s, a, y_i, z_j),$$

where α_w is the learning rate, and $Q = Q(s, a)$. The updates for the second layer's parameters are similar, but somewhat more involved. Again, from Equation (B.1),

$$\begin{aligned}
\Delta u_{i,j} &= (Q - R)\ \frac{\partial Q}{\partial u_{i,j}} \\
&= (Q - R)\ \frac{\partial}{\partial u_{i,j}}\ \vec{w} \cdot \vec{f} \\
&= (Q - R)\ \frac{\partial}{\partial u_{i,j}}\ \vec{w} \cdot \sum_k \vec{g}(\vec{x}, y_k)\ p(y_i \mid \vec{x}; \vec{u}_k) \\
&= (Q - R)\ \vec{w} \cdot \vec{g}(\vec{x}, y_i)\ \frac{\partial}{\partial u_{i,j}}\ p(y_i \mid \vec{x}; \vec{u}_i). \qquad \text{(B.2)}
\end{aligned}$$

Considering the final term in the above equation separately,

$$\begin{aligned}
\frac{\partial}{\partial u_{i,j}}\ p(y_i \mid \vec{x}; \vec{u}_i) &= \frac{\partial}{\partial u_{i,j}}\ \frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}, \qquad \text{where } Z = \sum_k e^{\vec{u}_k \cdot \vec{x}} \\
&= \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) \frac{\frac{\partial}{\partial u_{i,j}}\ \frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}}{\left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right)} \\
&= \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) \frac{\partial}{\partial u_{i,j}}\ \log\left[\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right] \\
&= \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) \left[x_j - \frac{\partial}{\partial u_{i,j}}\ \log Z\right] \\
&= \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) \left[x_j - \frac{1}{Z}\frac{\partial Z}{\partial u_{i,j}}\right] \\
&= \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) \left[x_j - \frac{1}{Z}\frac{\partial}{\partial u_{i,j}} \sum_k e^{\vec{u}_k \cdot \vec{x}}\right] \\
&= \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) \left[x_j - \frac{1}{Z}\ x_j e^{\vec{u}_k \cdot \vec{x}}\right] \\
&= \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) x_j \left[1 - \frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right].
\end{aligned}$$

Therefore, from Equation (B.2),

$$\begin{aligned}
\Delta u_{i,j} &= (Q - R)\ \vec{w} \cdot \vec{g}(\vec{x}, y_i)\ \frac{\partial}{\partial u_{i,j}}\ p(y_i \mid \vec{x}; \vec{u}_i) \\
&= (Q - R)\ \vec{w} \cdot \vec{g}(\vec{x}, y_i)\ \left(\frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right) x_j\ \left[1 - \frac{e^{\vec{u}_i \cdot \vec{x}}}{Z}\right] \\
&= (Q - R)\ x_j\ \ \vec{w} \cdot \vec{g}(\vec{x}, y_i)\ p(y_i \mid \vec{x}; \vec{u}_i)\ [1 - p(y_i \mid \vec{x}; \vec{u}_i)] \\
&= (Q - R)\ x_j\ \hat{Q}\ \left[1 - p(y_i \mid \vec{x}; \vec{u}_i)\right],
\end{aligned}$$

$$\text{where } \hat{Q} = \vec{w} \cdot \vec{g}(\vec{x}, y_i)\ p(y_i \mid \vec{x}; \vec{u}_i).$$

The resulting online updates for the sentence relevance and predicate labeling parameters $\vec{u}$ and $\vec{v}$ are:

$$\vec{u}_i \leftarrow \vec{u}_i + \alpha_u\ [Q - R(s_\tau)]\ \hat{Q}\ \vec{x}\ [1 - p(y_i|\cdot)],$$

$$\vec{v}_i \leftarrow \vec{v}_i + \alpha_v\ [Q - R(s_\tau)]\ \hat{Q}\ \vec{x}\ [1 - p(z_i|\cdot)].$$

B.2 Example of Sentence Relevance Predictions

Shown below is a portion of the strategy guide for Civilization II. Sentences that were identified as relevant by our text-aware model are highlighted in green.

Choosing your location.
When building a new city, carefully plan where you place it. Citizens can work the terrain surrounding the city square in an x-shaped pattern (see city radius for a diagram showing the exact dimensions). This area is called the city radius (the terrain square on which the settlers were standing becomes the city square). The natural resources available where a population settles affect its ability to produce food and goods. Cities built on or near water sources can irrigate to increase their crop yields, and cities near mineral outcroppings can mine for raw materials. On the other hand, cities surrounded by desert are always handicapped by the aridness of their terrain, and cities encircled by mountains find arable cropland at a premium. In addition to the economic potential within the city's radius, you need to consider the proximity of other cities and the strategic value of a location. Ideally, you want to locate cities in areas that offer a combination of benefits : food for population growth, raw materials for production, and river or coastal areas for trade. Where possible, take advantage of the presence of special resources on terrain squares (see terrain & movement for details on their benefits).

Strategic value.
The strategic value of a city site is a final consideration. A city square's underlying terrain can increase any defender's strength when that city comes under attack. In some circumstances, the defensive value of a particular city's terrain might be more important than the economic value; consider the case where a continent narrows to a bottleneck and a rival holds the other side. Good defensive terrain (hills, mountains, and jungle) is generally poor for food production and inhibits the early growth of a city. If you need to compromise between growth and defense, build the city on a plains or grassland square with a river running through it if possible. This yields decent trade production and gains a 50 percent defense bonus. Regardless of where a city is built, the city square is easier to defend than the same unimproved terrain. In a city you can build the city walls improvement, which triples the defense factors of military units stationed there. Also, units defending a city square are destroyed one at a time if they lose. Outside of cities, all units stacked together are destroyed when any military unit in the stack is defeated (units in fortresses are the only exception; see fortresses). Placing some cities on the seacoast gives you access to the ocean. You can launch ship units to explore the world and to transport your units overseas. With few coastal cities, your sea power is inhibited.

B.3 Examples of Predicate Labeling Predictions

Listed below are the predicate labelings computed by our text-aware method on example sentences from the game manual. The predicted labels are indicated below the words with the letters A, S, and B for *action-description*, *state-description* and *background* respectively. Incorrect labels are indicated by a red check mark, along with the correct label in brackets.

- After the road (S) is (S) built (S), use (A) the settlers (A) to start (A) improving (A) the terrain (A).
- When the settlers (S) becomes (S) active (S), chose (A) build (A) road (A).
- Use (A) settlers (✗ S(A)) or engineers (A) to improve (A) a terrain (S) square (✗ A(S)) within (S) the (S) city (S) radius (S)
- Bronze (S) working (S) allows (✗ B(S)) you to build (A) phalanx (A) units (A)
- In (S) order (S) to expand (✗ A(S)) your civilization (S), you need (B) to build (A) other cities (✗ B(A))
- In (✗ B(S)) order (S) to protect (✗ B(S)) the city, the phalanx (A) must (✗ S(A)) remain (A) inside (✗ B(A))
- As (✗ B(S)) soon as you've found (S) a (✗ B(S)) decent site (S), you want (✗ A(B)) your settlers (✗ B(A)) to build (A) a permanent (✗ S(A)) settlement - a city (A)
- In a city (✗ A(S)) you can build (✗ B(A)) the city (A) walls (A) improvement (A)
- Once (S) the city (S) is undefended (✗ B(S)), you can move (A) a friendly (A) army (A) into the city (A) and capture it
- You can build (A) a city (✗ S(A)) on any terrain (✗ B(S)) square (S) except (✗ A(S)) for ocean (S).
- You can launch (A) ship (A) units (✗ S(A)) to explore (S) the world (✗ B(S)) and to transport (✗ B(S)) your units (S) overseas (B)
- When a city (✗ A(S)) is in disorder (S), disband (A) distant military (A) units (✗ S(A)), return (A) them to their home (A) cities (✗ S(A)), or change (A) their home (A) cities (A)
- You can build (A) a wonder (✗ S(A)) only if you have discovered (S) the advance (S) that makes (S) it possible (S)

B.4 Examples of Learned Text to Game Attribute Mappings

Shown below are examples of some of the word to game-attribute associations learnt by our model. The top ten game attributes with the strongest association by feature weight are listed for three of the example words – "attack", "build" and "grassland". For the fourth word, "settler", only seven attributes had non-zero weights in experiments used to collect these statistics.

attack

- phalanx *(unit)*
- warriors *(unit)*
- colossus *(wonder)*
- city walls *(city improvement)*
- archers *(unit)*
- catapult *(unit)*
- palace *(city improvement)*
- coinage *(city production)*
- city_build_warriors *(action)*
- city_build_phalanx *(action)*

build

- worker_goto *(action)*
- settler_autosettle *(action)*
- worker_autosettle *(action)*
- pheasant *(terrain attribute)*
- settler_irrigate *(action)*
- worker_mine *(action)*
- build_city_walls *(action)*
- build_catapult *(action)*
- swamp *(terrain attribute)*
- grassland *(terrain attribute)*

grassland

- settler_build_city *(action)*
- worker_continue_action *(action)*
- pheasant *(terrain attribute)*
- city_build_improvement *(action)*
- city_max_production *(action)*
- settlers *(state attribute)*
- city_max_food *(action)*
- settler_goto *(action)*
- worker_build_road *(action)*
- pyramids *(city attribute)*

settler

- settlers *(state attribute)*
- settler_build_city *(action)*
- city *(state_attribute)*
- grassland *(terrain_attribute)*
- plains *(terrain_attribute)*
- road *(terrain_attribute)*
- workers *(state attribute)*

B.5 Features

Features used predict sentence relevance

The following templates are used to compute the features for sentence relevance:

- *Word W is present in sentence.*
- *Number of words that match the text label of the current unit, an attribute in the immediate neighbourhood of the unit, or the action under consideration.*
- *The unit's type is U, (e.g.,* `worker`*) and word W is present in sentence.*
- *The action type is A, (e.g.,* `irrigate`*) and word W is present in sentence.*

Features used predict predicate structure

The following templates are used to compute the features for the predicate labeling of words. The label being considered for the word (i.e., `action`, `state` or `background`) is denoted by L.

- *Label is L and the word type is W.*
- *Label is L and the part-of-speech tag of the word is T.*
- *Label is L and the parent word in the dependency tree is W.*
- *Label is L and the dependency type to the dependency parent word is D.*
- *Label is L and the part-of-speech of the dependency parent word is T.*
- *Label is L and the word is a leaf node in the dependency tree.*
- *Label is L and the word is not a leaf node in the dependency tree.*
- *Label is L and the word matches a state attribute name.*
- *Label is L and the word matches a unit type name.*
- *Label is L and the word matches a action name.*

Features used to model action-value function

The following templates are used to compute the features of the action-value approximation. Unless otherwise mentioned, the features look at the attributes of the player controlled by our model.

- *Percentage of world controlled.*
- *Percentage of world explored.*
- *Player's game score.*
- *Opponent's game score.*
- *Number of cities.*
- *Average size of cities.*
- *Total size of cities.*
- *Number of units.*
- *Number of veteran units.*
- *Wealth in gold.*
- *Excess food produced.*
- *Excess shield produced.*
- *Excess trade produced.*
- *Excess science produced.*
- *Excess gold produced.*
- *Excess luxury produced.*
- *Name of technology currently being researched.*
- *Percentage completion of current research.*
- *Percentage remaining of current research.*
- *Number of game turns before current research is completed.*

The following feature templates are applied to each city controlled by the player:

- *Current size of city.*
- *Number of turns before city grows in size.*
- *Amount of food stored in city.*
- *Amount of shield stored in city ("shields" are used to construct new buildings and units in the city).*
- *Turns remaining before current construction is completed.*
- *Surplus food production in city.*
- *Surplus shield production in city.*
- *Surplus trade production in city.*
- *Surplus science production in city.*
- *Surplus gold production in city.*
- *Surplus luxury production in city.*
- *Distance to closest friendly city.*
- *Average distance to friendly cities.*
- *City governance type.*
- *Type of building or unit currently under construction.*
- *Types of buildings already constructed in city.*
- *Type of terrain surrounding the city.*
- *Type of resources available in the city's neighbourhood.*
- *Is there another city in the neighbourhood.*
- *Is there an enemy unit in the neighbourhood.*
- *Is there an enemy city in the neighbourhood.*

The following feature templates are applied to each unit controlled by the player:

- *Type of unit.*
- *Moves left for unit in current game turn.*
- *Current health of unit.*
- *Hit-points of unit.*
- *Is unit a veteran.*
- *Distance to closest friendly city.*
- *Average distance to friendly cities.*
- *Type of terrain surrounding the unit.*
- *Type of resources available in the unit's neighbourhood.*
- *Is there an enemy unit in the neighbourhood.*
- *Is there an enemy city in the neighbourhood.*

The following feature templates are applied to each predicate-labeled word in the sentence selected as relevant, combined with the current state and action attributes:

- *Word W is present in sentence, and the action being considered is A.*
- *Word W with predicate label P is present in sentence, and the action being considered is A.*
- *Word W is present in sentence, the current unit's type is U, and the action being considered is A.*
- *Word W with predicate label P is present in sentence, the current unit's type is U, and the action being considered is A.*
- *Word W is present in sentence, and the current unit's type is U.*
- *Word W with predicate label P is present in sentence, and the current unit's type is U.*
- *Word W is present in sentence, and an attribute with text label A is present in the current unit's neighbourhood.*

- *Word W with predicate label P is present in sentence, and an attribute with text label A is present in the current unit's neighbourhood.*

Appendix C

High-level Planning

C.1 Features

Features used to predict preconditions from text

Given a sentence containing two words A and B that each describe a subgoal, the following templates are used to compute features on the dependency path between A and B. Two different features are computed depending on whether A occurs before B in the sentence, or vice versa. To aid generalization, words A and B themselves replaced with the token "OBJ_WORD".

- *Word W is present in A-B path.*
- *Dependency type D is present in A-B path.*
- *Word W is present in A-B path, and has dependency of type D to some other word.*

Features used to predict subgoal sequence

The following templates were used to compute the features used for subgoal sequence prediction:

- *Subgoal is reachable (subgoal was reached by some previous plan).*
- *Subgoal is predicted precondition for next subgoal*

- *Subgoal is predicted precondition for next subgoal, and is reachable*
- *Subgoal is not a predicted precondition for next subgoal*
- *Every pair of subgoal predicates. E.g.,* `(> (have wood) 0) :: (> (have stone) 0)`
- *Every pair of subgoal predicate parameter values. E.g.,* `wood :: stone`

Bibliography

[1] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intell.*, 71(1):43–100, 1994.

[2] R. Balla and A. Fern. UCT for tactical assault planning in real-time strategy games. In *21st International Joint Conference on Artificial Intelligence*, 2009.

[3] Kobus Barnard and David A. Forsyth. Learning the semantics of words and pictures. In *Proceedings of ICCV*, 2001.

[4] Jennifer L. Barry, Leslie Pack Kaelbling, and Toms Lozano-Prez. DetH*: Approximate hierarchical solution of large markov decision processes. In *IJCAI'11*, pages 1928–1935, 2011.

[5] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13:341–379, October 2003.

[6] Brandon Beamer and Roxana Girju. Using a bigram event model to predict causal potential. In *Proceedings of CICLing*, pages 430–441, 2009.

[7] Darse Billings, Lourdes Peña Castillo, Jonathan Schaeffer, and Duane Szafron. Using probabilistic knowledge and simulation to play poker. In *16th National Conference on Artificial Intelligence*, pages 697–703, 1999.

[8] Eduardo Blanco, Nuria Castell, and Dan Moldovan. Causal relation extraction. In *Proceedings of the LREC'08*, 2008.

[9] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in NIPS*, pages 369–376, 1995.

[10] S.R.K Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of ACL*, pages 82–90, 2009.

[11] S.R.K Branavan, Luke Zettlemoyer, and Regina Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *Proceedings of ACL*, pages 1268–1277, 2010.

[12] S.R.K. Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a monte-carlo framework. In *Proceedings of ACL*, pages 268–277, 2011.

[13] S.R.K. Branavan, David Silver, and Regina Barzilay. Non-linear monte-carlo search in civilization ii. In *Proceedings of IJCAI*, 2011.

[14] John S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In *Advances in NIPS*, pages 211–217, 1990.

[15] Arthur E. Bryson and Yu-Chi Ho. *Applied optimal control: optimization, estimation, and control.* Blaisdell Publishing Company, 1969.

[16] Du-Seong Chang and Key-Sun Choi. Incremental cue phrase learning and bootstrapping method for causality extraction using cue phrase and word pair probabilities. *Inf. Process. Manage.*, 42(3):662–678, 2006.

[17] David L. Chen and Raymond J. Mooney. Learning to sportscast: a test of grounded language acquisition. In *Proceedings of ICML*, 2008.

[18] David L. Chen and Raymond J. Mooney. Learning to interpret natural language navigation instructions from observations. In *Proceedings of AAAI*, pages 859–865, 2011.

[19] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of CoNNL*, pages 18–27, 2010.

[20] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of CoNNL*, pages 18–27, 2010.

[21] Christian Darken and John Moody. Note on learning rate schedules for stochastic optimization. In *Advances in NIPS*, pages 832–838, 1990.

[22] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC 2006*, 2006.

[23] Stephen Della Pietra, Vincent J. Della Pietra, and John D. Lafferty. Inducing features of random fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(4):380–393, 1997.

[24] Barbara Di Eugenio. Understanding natural language instructions: the case of purpose clauses. In *Proceedings of ACL*, pages 120–127, 1992.

[25] Q. Do, Y. Chan, and D. Roth. Minimally supervised event causality identification. In *EMNLP*, 7 2011.

[26] Jacob Eisenstein, James Clarke, Dan Goldwasser, and Dan Roth. Reading to learn: Constructing features from semantic abstracts. In *Proceedings of EMNLP*, pages 958–967, 2009.

[27] Michael Fleischman and Deb Roy. Intentional context in situated natural language learning. In *Proceedings of CoNLL*, pages 104–111, 2005.

[28] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:2003, 2003.

[29] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.

[30] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Morgan Kaufmann, 2004.

[31] Roxana Girju and Dan I. Moldovan. Text mining for causal relations. In *Proceedigns of FLAIRS*, pages 360–364, 2002.

[32] Dan Goldwasser and Dan Roth. Learning from natural instructions. In *Proceedings of IJCAI*, pages 1794–1800, 2011.

[33] Dan Goldwasser, Roi Reichart, James Clarke, and Dan Roth. Confidence driven unsupervised semantic parsing. In *Proceedings of ACL*, pages 1486–1495, 2011.

[34] Peter Gorniak and Deb Roy. Speaking with your sidekick: Understanding situated speech in computer role playing games. In *Proceedings of AAAI*, 2005.

[35] Geoffrey Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14:1771–1800, 2000.

[36] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[37] Nicholas K. Jong and Peter Stone. Model-based function approximation in reinforcement learning. In *Proceedings of AAMAS*, pages 670–677, 2007.

[38] Anders Jonsson and Andrew Barto. A causal approach to hierarchical decomposition of factored mdps. In *Advances in Neural Information Processing Systems, 13:10541060*, page 22. Press, 2005.

[39] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of ICML*, 2001.

[40] Tessa Lau, Clemens Drews, and Jeffrey Nichols. Interpreting written how-to instructions. In *Proceedings of IJCAI*, pages 1433–1438, 2009.

[41] Marián Lekavý and Pavol Návrat. Expressivity of strips-like and htn-like planning. *Lecture Notes in Artificial Intelligence*, 4496:121–130, 2007.

[42] Oliver Lemon and Ioannis Konstas. User simulations for context-sensitive speech recognition in spoken dialogue systems. In *Proceedings of EACL*, pages 505–513, 2009.

[43] Percy Liang, Michael I. Jordan, and Dan Klein. Learning semantic correspondences with less supervision. In *Proceedings of ACL*, pages 91–99, 2009.

[44] Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. In *Proceedings of ACL*, pages 590–599, 2011.

[45] Diane J. Litman, Michael S. Kearns, Satinder Singh, and Marilyn A. Walker. Automatic optimization of dialogue management. In *Proceedings of COLING*, 2000.

[46] Matt MacMahon, Brian Stankiewicz, and Benjamin Kuipers. Walk the talk: connecting language, knowledge, and action in route instructions. In *Proceedings of AAAI*, pages 1475–1482, 2006.

[47] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[48] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. Automatic discovery and transfer of maxq hierarchies. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 648–655, 2008.

[49] Raymond J. Mooney. Learning to connect language and perception. In *Proceedings of AAAI*, pages 1598–1601, 2008.

[50] Raymond J. Mooney. Learning language from its perceptual context. In *Proceedings of ECML/PKDD*, 2008.

[51] A. Newell, J.C. Shaw, and H.A. Simon. *The processes of creative thinking.* Paper P-1320. Rand Corporation, 1959.

[52] James Timothy Oates. *Grounding knowledge in sensors: Unsupervised learning for language and planning.* PhD thesis, University of Massachusetts Amherst, 2001.

[53] Warren B Powell. *Approximate Dynamic Programming.* Wiley-Interscience, 2007.

[54] Deb K. Roy and Alex P. Pentland. Learning words from sights and sounds: a computational model. *Cognitive Science 26*, pages 113–146, 2002.

[55] Nicholas Roy, Joelle Pineau, and Sebastian Thrun. Spoken dialogue management using probabilistic reasoning. In *Proceedings of ACL*, 2000.

[56] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[57] J. Schäfer. The UCT algorithm applied to games with imperfect information. Diploma Thesis. Otto-von-Guericke-Universität Magdeburg, 2008.

[58] Jost Schatzmann and Steve Young. The hidden agenda user simulation model. *IEEE Trans. Audio, Speech and Language Processing*, 17(4):733–747, 2009.

[59] Konrad Scheffler and Steve Young. Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning. In *Proceedings of HLT*, 2002.

[60] B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134 (1-2):241–275, 2002.

[61] Avirup Sil and Alexander Yates. Extracting STRIPS representations of actions and events. In *Recent Advances in Natural Language Learning (RANLP)*, 2011.

[62] Avirup Sil, Fei Huang, and Alexander Yates. Extracting action and event semantics from web text. In *AAAI 2010 Fall Symposium on Commonsense Knowledge (CSK)*, 2010.

[63] D. Silver, R. Sutton, and M. Müller. Sample-based learning and search with permanent and transient memories. In *25th International Conference on Machine Learning*, pages 968–975, 2008.

[64] Satinder Singh, Diane Litman, Michael Kearns, and Marilyn Walker. Optimizing dialogue management with reinforcement learning: Experiments with the njfun system. *Journal of Artificial Intelligence Research*, 16:105–133, 2002.

[65] Satinder P. Singh, Michael J. Kearns, Diane J. Litman, and Marilyn A. Walker. Reinforcement learning for spoken dialogue systems. In *Advances in NIPS*, 1999.

[66] Jeffrey Mark Siskind. Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *Journal of Artificial Intelligence Research*, 15:31–90, 2001.

[67] N. Sturtevant. An analysis of UCT in multi-player games. In *6th International Conference on Computers and Games*, pages 37–49, 2008.

[68] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 1998.

[69] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in NIPS*, pages 1057–1063, 2000.

[70] Richard S. Sutton, Anna Koop, and David Silver. On the role of tracking in stationary environments. In *Proceedings of ICML*, pages 871–878, 2007.

[71] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R. Walter, Ashis Gopal Banerjee, Seth Teller, and Nicholas Roy. Understaning natural language commands for robotic navigation and mobile manipulation. In *Proceedings of AAAI*, 2011.

[72] G. Tesauro and G. Galperin. On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing 9*, pages 1068–1074, 1996.

[73] Paulina Varshavskaya, Leslie Pack Kaelbling, and Daniela Rus. Automated design of adaptive controllers for modular robots using reinforcement learning. *The International Journal of Robotics Research*, 27:505–526, 2008.

[74] Adam Vogel and Daniel Jurafsky. Learning to follow navigational directions. In *Proceedings of the ACL*, pages 806–814, 2010.

[75] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

[76] Terry Winograd. *Understanding Natural Language.* Academic Press, 1972.

[77] Alicia P. Wolfe and Andrew G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *In Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 816–823, 2005.

[78] Yuk Wah Wong and Raymond J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *ACL*, 2007.

[79] Chen Yu and Dana H. Ballard. On the integration of grounding language and learning objects. In *Proceedings of AAAI*, pages 488–493, 2004.

[80] Luke Zettlemoyer and Michael Collins. Learning context-dependent mappings from sentences to logical form. In *Proceedings of ACL*, pages 976–984, 2009.